## NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

## AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

# Hierarchical Design of Delay-Insensitive Systems Using Fine-Grain Building Blocks

Ping Ngai Lam

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

March 1989

ISBN 0-315-49097-7

## ABSTRACT

Hierarchical Design of Delay-Insensitive Systems
Using Fine-Grain Building Blocks

Ping N. Lam

As VLSI systems grow larger as a result of the constant downscaling of circuit dimensions, the use of global clock signals for system timing may no longer be appropriate. There is a growing interest toward more asynchronous, distributed types of architectures. One of the motivating factors is that the propagation delays of wires do not scale down proportionally with the switching times of the gates. Increased wire delays are limiting the performance of the system and are a source for higher design complexity. One method of dealing with this problem is with the use of asynchronous, delay-insensitive circuits, where the correctness of operation is not affected by either interconnecting wire or circuit module delays. This thesis presents a set of fine-grain delay-insensitive (DI) building-blocks which are meant to be used as the basic building blocks in a delay-insensitive silicon compiler. These blocks are formally specified using an extended STG (Signal Transition Graph) model which allows DI modules and delay-sensitivity to be clearly specified. A hierarchical composition procedure is described for composing deterministic STG specifications. Through the use of a packet model, two example circuits are designed.

## ACKNOWLEDGEMENTS

I would like to thank Dr. H.F. Li, my thesis supervisor for all his help, understanding, and guidance. His constructive criticisms and keen ability to pinpoint important issues and weaknesses has been a major part in the shaping of this thesis.

I would also like to thank the present and former members of the DI group, including Dr. D.K. Probst for spicing up the meetings, Dr. R. Jayakumar, S.C. Leung, and R. Prasad. Thanks also goes to M. Zelada for help in using Ventura.

I would also like to thank my family, including my parents, brother, and sister for putting up with me for so long.

This research was funded by an NSERC scholarship and a CRIM bursary. The preparation of this document was done using Ventura and the equipment at Concordia University.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

There is an increasing interest in the use of asynchronous design techniques for building large VLSI systems, moving away from the totally synchronous, globally clocked method of system timing. One of the factors motivating this change is the problems that are faced as VLSI circuits are scaled down. Wire delays become increasingly significant relative to gate delays, thereby dominating system performance and increasing the design complexity. Research into asynchronous techniques is centered around self-timed, speed-independent, and delay-insensitive systems. This thesis will adopt the use of delay-insensitive circuits because of its potential for lower design complexity in the construction of very large systems.

## 1.1 Motivation

The majority of the presently existing VLSI circuits are synchronous and operate using a single global clock. Every part of the circuit operates in step with this global signal completing a computation within each clock cycle. Although synchronous clocking has served as an appropriate and efficient method of timing, there are several important reasons for moving away from totally synchronous systems to asynchronous systems. In the area of asynchronous systems, the use of delay-insensitive circuits are advocated. The main reasons are

1) problem with scaling
2) speed
3) design complexity
4) reliability

The most important and immediately obvious reason is the problem that occurs as a result of scaling of VLSI circuits. As circuits are scaled down, the

propagation delays of wires do not scale proportionately to that of the switching times of the gates. Interconnection delays become increasingly dominant in a circuit [Sara82], and hence it is becoming more difficult to route long wires without experiencing delay problems. At the same time, as systems become larger as a result of scaling, global signals must traverse a proportionately longer distance. The original assumption of a "simultaneous" global clock signal is no longer valid, and much effort must go into the proper distribution of a clock and the prevention of clock skews [Wagn88].

The fact that wire delays do not scale proportionately to gate delays has been shown in [Seit80]. When circuit dimensions are scaled down by a factor of $k$, gate switching time decreases approximately by $k$. For wires, scaling down by a factor of $k$ results in a total increase in resistance by $k$; a decrease in the length and the width of a wire results in no change in resistance, but a decrease in the thickness of the wire (3 dimensionally) to maintain relative surface flatness results in an increase by $k$. The overall capacitance of the wire scaled down by a factor of $k$ decreases by $k$; a decrease in the length and width of the wire results in a quadratic decrease in capacitance, but the use of thinner oxides results in an increase in capacitance by $k$. The overall delay of the scaled wire therefore does not change, since the increase in resistance by $k$ and decrease in capacitance by $k$ cancel each other out (this is assuming that the resistance and parasitic capacitance dominate the wire signal propagation time on a chip, rather than the speed-of-light delay; the speed-of-light delay will have to be taken into account when gate speeds increase further).

It can be seen that as VLSI circuits are scaled down, the wire delays become increasingly significant relative to the gate delays. Inserting drivers on long stretches of wires reduces, but does not eliminate the problem. Note that the use of a larger driver with a lower impedance will not be able to decrease the signal propagation time significantly because of the increased resistance of the wire itself. As a system becomes larger due to scaling, a proportionately larger load (fanout) is placed on the system clock. The consequence is that global

clocks become a major performance bottleneck of the system. The fraction of time with which a clock pulse travels to all parts of the circuit will become significantly greater than the switching times of the clocked circuits. In addition, the clock delay must be further increased to accommodate for clock skew. This results in the system being idle most of the time, having to wait for a slow global clock signal (slow relative to the increased speed of the gates).

The use of asynchronous circuits rather than synchronous circuits can increase the speed of the system. The individual parts of the circuit do not have to wait for a global clock signal to tell them when to proceed with the next operation. Circuits can proceed as soon as they are ready and as soon as data is available from their neighbors. One property of clocked systems is that they operate on worst case delays. The width of the clock pulse must be long enough to allow for the slowest part of the circuit to operate. In addition, an error margin must be included in estimating the delay of this critical path. In a large system, much of the circuits can often operate faster than this worst case delay. Asynchronous systems on the other hand operate on average case delays. The speed of operation is dependent on the input data and the speed of its neighbors.

The use of asynchronous circuits in place of synchronous ones may solve some problems, but introduce some new ones as well. The main problem is with the variation in gate and wire delays as a result of the manufacturing process, scaling, temperature variations, and the actual layout. Clocked systems allow for variation in gate delays because the clock synchronizes all operations to occur in step, thereby hiding the differences in gate delays within the duration of a clock pulse. If a strategy is not used to manage these differences in delays, the design complexity of the asynchronous system will become intractable.

The use of delay-insensitive circuits is therefore advocated to manage the design complexity. Delay-insensitive circuits operate correctly regardless of both the circuit (module) and wire delays. Its use allows for the important separation of the physical timing issues from the algorithmic, logical correctness of the system. The correctness of the system is independent of the lower level

delays. This independence from the physical implementation details of the circuit is an important property in reducing the design complexity of the system.

The best method to manage complexity is with the use of hierarchy, partitioning the system into levels of individual modules; the top level system is a single highest level module, which is composed of smaller lower level modules, which are in turn composed of further lower level modules. The division into different levels of hierarchy allows for a "divide-and-conquer" strategy in managing a complex system. The essential requirement for the individual modules in the hierarchy is that they have clearly defined interfaces so that the modules may be treated locally and independently of one another. A change in one module should not affect the correct operation of another module, as long as the interface specification for that module remains the same. A variation in delay of one module should not affect the correctness of the rest of the system. Any delays resulting from the interconnection medium should not affect the correctness either. The use of modules with clean interfaces is possible in delay-insensitive systems

Clocked systems on the other hand do not have well defined interfaces. This is because the clock signal crosses the boundary of every level of the hierarchy, down to the lowest level module. A change in the delay of the clock line as a result of scaling or differences in layout of the modules may drastically affect the correctness of operation An increase in the delay of one part of the circuit (for example, the part containing a critical path) may require the adjustment of the clock, which will in turn affect every other part of the system.

The last argument for the use of asynchronous systems is reliability. This is caused by the possibility of metastability in circuits [Klee87, Mari81, Chan73]. Metastability refers to a circuit lingering between two stable states for an indefinite period of time as a result of marginal triggering. An example of metastable operation is at the input of a clocked system which must sample asynchronous data. I/O devices generally are asynchronous and must be interfaced with synchronous systems. Metastability can occur whenever the clock latches in

asynchronous data which is in the middle of changing values. Since clocked systems require all computations to complete within a bounded period of time, the possibility of metastability occurring for an unbounded period can lead to malfunctioning. As system clock speeds increase, there is an increasingly higher probability of metastability occurring. Another example of metastable operation is in arbiters. Two simultaneous requests will cause the arbiter to enter a metastable state; resolution occurs only after an indefinite period of time. The use of asynchronous systems can solve the reliability problem because asynchronous circuits do not require a fixed bounded period of time for an operation to occur. Asynchronous circuits can interface directly with asynchronous inputs and computation resumes as soon as both parties are ready.

## 1.2 Asynchronous Circuits

The traditional, earlier form of asynchronous circuits are commonly known as Huffman asynchronous sequential circuits [Unge69]. Such a circuit consists of a combinational circuit which has some of its inputs coming from the environment and some of its inputs connected to its own outputs in the form of feedback paths. These circuits are normally designed using a finite state machine model and flow tables Huffman asynchronous circuits are not in common use because of the great difficulty in designing them. The reason is because of the many races and hazards that may occur to cause the circuit to malfunction. These hazards are due to variations in gate delay, especially in the feedback paths of the circuit. These delays are very difficult to control. Both upper and lower bound delays must be placed on the gates in order for correct operation. The finite state machine model used in designing such circuits does not model concurrency, even though these asynchronous circuits are concurrent and are continuously accepting input changes. As circuits are scaled down, even more problems appear as a result of increased wire delays. It is obvious that these types of circuits are not practical in the design of all but the simplest types

of circuits. In order to manage the exponential design complexity of asynchronous systems, different design techniques are required.

## 1.3 Self-Timed Circuits

Self-timed circuits [Seit80] are circuits which are timed locally and do not have a global clock. Elements in a self-timed system communicate locally using initiation and completion signals, and the sequencing of computational events is determined by the way the individual elements are connected together. The time required to complete a computation is determined by the delays of the elements and their communication delays. Communication between two elements is implemented using a self-timed signalling protocol, such as 2-phase or 4-phase handshaking. The use of this type of communication protocol allows a circuit to operate correctly independent of the speeds with which the elements operate or their interconnection delays. This is referred to as delay-insensitive communication.

In the design of the individual elements of a self-timed system, there can be some delay assumptions made. One such assumption is that of equipotential regions. An equipotential region is a region in which a signal on a wire is treated as identical at all points on the wire within that region. This negligible wire delay assumption is valid within small regions and as long as the wire delays are small compared to the switching times of the gates.

There are different possibilities in the implementation of self-timed systems. The type which has been used by Seitz are classified as globally-asynchronous locally-synchronous systems [Chap84]. Globally, communication is asynchronous with the use of completion signals, and locally, the elements run on a local clock. A stretchable clock is used to handle the problem of metastability; it stops synchronously and restarts asynchronously. The communication protocol used by the elements in this type of system is not totally insensitive to delays. The interface of the elements consist of a

request/acknowledge wire pair and a set of data lines. The delay assumption required in this type of system is that the data lines have a shorter delay than the request line so that the data is available at the destination element before the request signal enters and requests processing of the data.

Another possible implementation of self-timed systems is used in [Dall87] where the delay-insensitivity is relaxed even further. It is similar to that used by Seitz, except that bidirectional lines are used for communication; because the request/acknowledge lines are combined into a single wire, even more delay assumptions must be made.

Alternatively, the delay-insensitivity of the elements can be strengthened. Communication can be based solely on request/acknowledge lines using a protocol so that communication is strictly delay-insensitive. Self-timed circuits may also be completely clock-free. Speed-independent circuits are an example of totally clock-free circuits. The name self-timed has been used for different sub-classes of circuits. It is a "loose" term used for naming circuits which use local timing rather than a global clock. Sub-classes of self-timed systems may either use local clocks or be clock-free and may have different degrees of delay-insensitivity.

## 1.4 Speed-Independent and Delay-Insensitive Circuits

Speed-independent [Chu87/87b, Mart86/86b/85, Bert88, Kell74, Mill65] and delay-insensitive circuits [Rose88, Prob88, Lam88/88b/89, Uddi86/84, Blac86, Moln85] are sub-classes of self-timed circuits. Speed-independent circuits have an earlier existence and refer to totally clock-free asynchronous circuits. Speed-independent circuits operate correctly regardless of the variations in delay of the logic gates. The basic elements or operators are gates and single wires are used for interconnection. Since wires are assumed to have negligible delays and an isochronic fork assumption must be made, a speed-independent system must exist within an equipotential region.

Delay-insensitive circuits, on the other hand, make no assumptions on wire delays. Delay-insensitive systems are partitioned into modules, called DI modules, and all module communication is protocol based to ensure strict delay-insensitivity. The correct operation of a delay-insensitive system is therefore independent of

(1) wire interconnection delays between modules
(2) computational delays internal to each module

The delay-insensitive system is composed out of a hierarchy of DI modules, and the lowest level DI module is called an atomic DI module. The interface of all DI modules must be strictly delay-insensitive, which is ensured by the use of a protocol based delay-insensitive communication. Although the higher level DI modules must be composed of lower level DI modules, the implementation of the atomic DI modules are not required to be delay-insensitive. For example, the atomic DI modules may have a speed-independent implementation, a delay-sensitive asynchronous circuit implementation, or even make use of a local clock. As long as the DI interface specification of the module is satisfied, we are not concerned with its particular implementation.

Note that the term delay-insensitivity has been used interchangeably with speed-independent and self-timed, without making a clear distinction between the classes of circuits. For example, Martin has called his circuits delay-insensitive in [Mart86], as well as self-timed. In [Dill88], the circuits are referred to as speed-independent. His circuits are actually speed-independent (under the class of general self-timed circuits) because isochronic fork assumptions are made.

Delay-insensitivity is mainly concerned with the composition of a large system, emphasizing the communication and interconnection of modules. Speed-independence addresses the lower level implementation of modules or the internal synthesis of a DI module. Note that speed-independent design cannot be used at the system level because the size of a speed-independent circuit cannot be arbitrarily large. Isochronic forks and wires must be contained

within an equipotential region, which is becoming smaller as scaling introduces greater wire delays.

The advantages of using delay-insensitive circuits were discussed previously. Its use allows for a clean interface for modules in a hierarchical design methodology. The correctness of the system is completely separated from the detailed timing issues, thereby lowering the design complexity. Since DI circuits operate correctly regardless of both the wire and module delays, there is freedom in the layout of the modules. This is a very important property if design automation is used, where it is difficult to place constraints on the length of wires in automated routing.

# Chapter 2

## Previous Work

The previous work done on asynchronous systems can be grouped into different categories. These work can be based upon the type of model used, and upon the emphasis of either analysis or synthesis of circuits. The earlier work done on asynchronous circuits has been based on the finite state machine model and the use of Huffman style asynchronous circuits. More recent work is based on speed-independent, delay-insensitive, and self-timed circuits. We will concentrate on surveying clock-free delay-insensitive and speed-independent circuits rather than general self-timed circuits. General self-timed circuits would also include locally clocked systems such as [Seitz80], as well as hand designed self-timed circuits

### 2.1 Trace Theory

Trace theory has been used as a model for specifying and analyzing delay-insensitive circuits [Uddi84/86, Blac86, Snep85, Rem83]. The behavior of a circuit is described by traces, which are sets of event sequences. These events are the signal transitions on the wires of a circuit; a signal transition can be a transition from high to low or low to high. Formally, a trace structure T is a pair $<aT, tT>$ where aT is the alphabet of T and tT is the set of traces of T. The alphabet is a set of symbols associated with the wires of the circuit. A trace is a sequence of alphabet symbols which describe the possible signal transition communication between the circuit module and its environment. The alphabet can be decomposed into an input and output alphabet, corresponding to the input and output wires of the module. Reasoning of circuits can then be done by defining and manipulating these trace structures. Operations on trace

structures such as projection, weave, and blend can be used to compose trace structures.

Udding has used trace theory to formally define and classify delay-insensitive circuits. Informally, delay-insensitivity can be described as a circuit which is free from transmission and computation interference. Transmission interference occurs when two signal transitions on the same wire interfere with one another. If two transitions come too close together, this results in the cancelling of the signals on the wire. In the trace structure T, this is specified by disallowing the existence of two consecutive identical alphabet symbols. Computation interference refers to a signal arriving when the receiving end is not ready for it. For example, before a module has finished processing a previous signal, a new input signal comes in. This new input causes interference in the module's previous computation because the module has not yet finished its previous computation and indicated its readiness with an output completion signal. Computation interference can also be caused by the delays on the wires; two signals travelling on two wires in the same direction, one after another, may have their order reversed. If the signals travel in opposite directions, their ordering should be immaterial also.

More recent work on trace theory has been done in [Dill88/88b]. Trace theory is used for the verification of speed-independent circuits. Given a circuit specification and an implementation, it is possible to verify whether the implementation meets that specification. The procedure was automated and had detected errors in previous circuit designs.

The main use of trace theory has been in analysis, rather than the synthesis of circuits. Trace theory is a fairly low level model. Its use in specifying and reasoning with concurrency is complex and difficult because trace theory is essentially a linear model based on linear traces. To specify a concurrent circuit, all the possible linear traces must be enumerated. This leads to a combinatorial explosion which requires the use of a computer. Manipulation and visualization

of traces are more difficult than that of higher level models. As well, it is not suited for describing properties such as choice and non-determinism.

## 2.2 Communicating Sequential Processes

Communicating Sequential Processes (CSP) was originally founded by Hoare [Hoare78]. It is a language which expresses concurrency by the use of processes communicating concurrently. It is based on guarded commands which is due to Dijkstra. There can be different versions of the original basic CSP language, including the version in [Hoar85].

CSP is used as a model for speed-independent circuit design by Martin [Mart85/86/86b/87]. A CSP-like language is used as the high level description of the circuit. Through a series of semantics-preserving transformations, the language is compiled into lower, gate level operators. The resulting circuit is correct by construction. The steps used in this compilation method is (1) process decomposition, (2) handshaking expansion, (3) reshuffling, (4) production rule expansion, and (5) operator reduction. Process decomposition is done on the original CSP program to reduce its size and to obtain guarded commands which can be directly translated. Handshaking expansion is done on these simplified commands; this means converting the two-wire communication channels into their 4-phase handshake protocol (essentially a linear trace of the four signal transitions implementing the handshake sequence). Reshuffling can then be done to reorder the linear sequence of handshake expansions. By reshuffling the second half of the handshake sequences, more area efficient circuits can be obtained. Production rule expansion transforms the sequence into a set of production rules. Operator reduction then converts these production rules to match the production rules of the operators.

This compilation method is done by hand, and there are some problems with it. The process is not algorithmic. There are some heuristics involved that are human guided, which results in an almost hand designed circuit. The problem

is mostly due to steps 4 and 5 Converting the handshake expansion into production rules requires the introduction of state variables. It is difficult to determine when and where to insert these state variables. In the operator reduction step, one essentially has to manipulate and "force" the production rules to fit into the production rules of the hardware template operators.

A more recent attempt has been made to automate this compilation procedure [Burn88]. The translation of the CSP program follows standard syntax-directed techniques. From the syntax of the defined language, the CSP program is transformed into corresponding circuit blocks, such as circuit blocks for sequencing, statements, and variables. Since the new compilation method does not use the reshuffling technique of the original method, the circuits may be less area efficient. For example, the switch process example shown in [Burn88] requires a lot of area and can be more simply implemented using channel demultiplexers.

CSP appears good as a basis for a source language for translating into VLSI circuits. This is because CSP closely resembles the nature of hardware circuits. Processes in CSP imply modularity, which is desirable in VLSI. Message passing communication is used in CSP, corresponding to circuit blocks directly communicating over wires Only local variables are used inside processes, and this is desirable if we do not want slow global wire signals on a VLSI chip. CSP appears more useful as a higher level description language than as a lower level specification model for concurrent circuit behavior.

The most recent use of CSP for silicon compilation is by the Philips group [Nies88]. As of this writing, they appear to be the most advanced in the area of compilation of self-timed/delay-insensitive systems.

## 2.3 Petri Nets

Petri nets [Pete81, Reis85] are a common tool for modelling systems which exhibit concurrency. They can be used in the design of a computer system, for

modelling an existing system, or used as an analysis tool. Petri nets are a graph based model which describe partial ordering and causal relationships between events. A Petri net structure can be described as < P, T, I, O > where P is a set of places, T is a set of transitions, and I and O are the functions for mapping input and output places to transitions. Transitions are interpreted as events or actions that occur, and places are interpreted as the conditions that may be true or false to enable an action to occur. The dynamic behavior of a Petri net is expressed by the use of tokens. A transition is enabled to fire if all of its input places contain a token. When a transition fires, a token is removed from each of its input places and a token is placed in each of its output places. There has been many tools and techniques developed in Petri net theory. In addition, there exist many classes, sub-classes and extensions to Petri nets. In the area of asynchronous circuits, Petri nets have also been used. This ranges from the modelling of asynchronous circuits to the direct implementation of Petri nets into the corresponding place/transition hardware. For speed-independent design, the most recent notable use of Petri nets is by Chu [Chu87/87b].

Chu uses Signal Transition Graphs (STG) as a specification model for the synthesis of speed-independent control circuits. STG's are interpreted Petri nets, which means that certain interpretations (semantics) are given to the places and transitions. STG's are directed graphs in which the vertices represent wire signal transitions and the arcs represent the causal relation between transitions. They have a direct correspondence with Petri nets: a Petri net transition corresponds to an STG vertex, and the Petri net places are implied on the STG arcs. The token firing sequence of an STG is the same as that for Petri nets.

The synthesis procedure used is (1) construct a preliminary STG specification according to the circuit description, (2) check the syntax of the STG for liveness and persistency, (3) decompose the STG using contraction, (4) convert the sub-graph STG's into state graphs, (5) obtain the logic function of the circuit from the state graphs. The preliminary STG specification of the control circuit must be done by hand Since this is error-prone, the STG must

be checked for liveness and persistency. Liveness ensures that the circuit does not contain deadlock. Essentially, this means making sure that the high and low signal transitions alternate. Persistency ensures that the circuit contains no hazards. A transition is called persistent if and only if when it is enabled the occurrence of some other transition does not disable it. This property is similar to that of computation interference; before the module is ready and has fired the (persistent) transition, another input transition occurs to disable it. Additional dependencies must be introduced to ensure this persistency, and possibly additional internal transitions as well. The correct STG can then be decomposed into smaller STG sub-graphs This is done by the use of contraction, the deletion of unwanted signal transitions. A sub-graph is obtained for each output signal of the desired circuit, and the signal transitions which do not affect this output is deleted via contraction. The decomposed STG sub-graphs are then directly translated into state graphs. The state graphs are then used to obtain the logic function of the sub-circuit

Algorithms are given such as for contraction, but the synthesis procedure requires some manual design. This is due mostly to the persistency requirements that must be met. This synthesis procedure is intended for building control modules and not for system level design. One problem that may occur is in the final step of the procedure where the state graph is transformed into logic circuits. Before the logic circuits are obtained, a K-map is used. Circuits obtained from K-maps such as C-elements, may contain hazards, which is shown in [Moln85].

The STG model is closely related to trace theory in that both use signal transition sequences to describe the behavior of an asynchronous circuit. STG's are a higher level model than trace theory because specifications are more concise due to the ability to explicitly describe concurrency. Trace theory would require the enumeration of all possible sets of traces to describe a concurrent circuit due to the linear nature of traces. Another model which is related to STG/Petri nets is partially ordered multisets (Pomsets) [Pratt82/86]. Current

work is being done [Prob88] using Pomsets for modelling delay-insensitive circuits.

## 2.4 Approach of this Thesis

The direction taken in this thesis is toward delay-insensitive systems. A basic set of building blocks, called DI building blocks, are provided for the construction of delay-insensitive circuits. These building blocks are fine-grained because they are nearly the smallest atomic modules possible that contain DI interfaces. The intention of these DI building blocks is to use them as the assembly language blocks in an automated silicon compiler (synonymous to assembly language statements in a software compiler). These blocks are similar to the speed-independent modules of [Kell74] but are delay-insensitive. The use of fine-grain building blocks allows for more flexibility in the layout of the circuit.

The basic communication medium used by these blocks are channels, the same as that used by Martin. This ensures that all modules have a common delay-insensitive interface. A packet model is used so that the design of delay-insensitive circuits can be done at the packet level of abstraction. The intention is to decrease the design complexity as much as possible in the building of VLSI circuits, moving away from the lower level details of wire signal transitions. The correctness of the circuit can then be reasoned out at a higher level. Two example circuits, a DI finite state machine and a circuit for distributed mutual exclusion, are shown to demonstrate how the building blocks are used.

The formal model used for the specification of the DI building blocks are STG's because of the clarity with which they express concurrent circuit behavior. Extensions are made to the basic model so that events controlled by the module and events controlled by the environment can be clearly distinguished. This is important in delay-insensitivity where modularity is required in order to express hierarchy. As well, delays are incorporated into the model, allowing for the specification of delay-sensitive parts of a circuit. Since partially delay-sensitive

hybrid building blocks are provided for the internal construction of the DI modules, the model must also be able to specify this. Thus, delay-insensitive circuits can be clearly distinguished from non-delay-insensitive circuits. A composition procedure is also described that can hierarchically compose deterministic STG specifications. It can obtain composite STG specifications which are otherwise difficult to obtain correctly by hand, such as the specification of FIFO elements with a buffer size greater than one. It has the potential for use in the verification of delay-insensitive circuits.

# Chapter 3

## Basis of the DI Modules

The basis of the DI modules that will be used in the construction of delay-insensitive circuits and systems will be the use of channels for communication. This type of communication will be delay insensitive, as long as the modules correctly handle the communication protocol. A system composed of a network of modules interconnected with channels will be a delay-insensitive system. In this chapter, an introduction to channels is given. Two basic building blocks are presented to sequentially manipulate channels. These building blocks, the P-block and the A-block, will serve as circuit examples for analysis done in the next chapter.

### 3.1 Channels

A channel is implemented with two wires, a request wire and an acknowledge wire. Given two processes (or modules), $P_1$ and $P_2$, the channel interface to $P_1$ would consist of a request wire $r_1$ and an acknowledge wire $a_1$; similarly, process $P_2$ would have channel interface wires $r_2$ and $a_2$. Processes $P_1$ and $P_2$ connected via a channel would have $r_1$ connected to $r_2$ and $a_1$ connected to $a_2$. A channel communication is specified by the trace $(r_1; r_2; a_2; a_1)^*$, where the asterisk denotes repetition of the sequence. Given that 4-phase handshaking is used, a single channel communication would consist of two of these cycles, the first cycle representing high wire transitions and the second cycle representing low wire transitions. In 2-phase handshaking, a single cycle represents a single channel signal, with consecutive channel signals alternating between the use of high and low wire transitions. Two-phase handshaking is also known as transition signalling or non-return to zero signalling, while four-phase handshaking is known as level signalling or return to zero signalling.

Depending upon the point of view with which a channel is seen, a channel can be referred to as:

1) an active output channel, or
2) a passive input channel.

The channel as seen from the point of view of process $P_1$ is an active output channel. In the stable state, $P_1$ initiates the first request $r_1$ to start the handshake sequence, and so is referred to as the *active* process sending an output channel signal. From the point of view of $P_2$, the channel is referred to as a passive input channel. In the stable state, a request $r_2$ comes into the process to initiate $P_2$ to complete its handshake sequence. $P_2$ is referred to as a *passive* process, accepting an input channel signal.

The stable state in the 4-phase handshake protocol is defined as the state in which both the request and acknowledge wires are low (seen by both $P_1$ and $P_2$). In the 2-phase handshake protocol, the stable state is defined as the state in which both the request and acknowledge wires are of the same value, either high or low. This refers to the local stable state of a channel. In an entire system which contains many processes communicating with many channels, the global stable state of the system refers to each of the internal local channels being in a stable state. In a global stable state, no signals may occur at the output ports of the system.

If the external interface of a process or a module consists solely of channels, and if the given module correctly handles these channel signals, it can be shown that this module's interface is delay-insensitive according to Udding's rules [Uddi84, 86]. Udding's classification of delay-insensitivity requires that a system be free of both transmission and computation interference. Modules communicating with channels are free from transmission interference. Any request signal sent from one module to another is guaranteed to arrive at the destination because the sender does not change its request wire until it has explicitly received an acknowledge signal. It is not possible to have two transitions on a wire interfere with one another, thereby cancelling each other out, because only one transition on a wire may occur at any time due to the use

of handshake signalling. Channel communication is also free of computation interference. The receiving module will always be ready to accept the next request because it has already acknowledged the previous request.

Because of the delay-insensitive nature of channels, the method with which we will build delay-insensitive systems is based on channels. In particular, the DI modules that will be used are based on *single* channels, a channel consisting of a single request wire and a single acknowledge wire. As will be seen, these basic single channels can be extended and converted to more general channels with different data encoding. The type of signalling that will be used is the 4-phase protocol. The advantage of this type of signalling is that it results in simpler circuit implementations as compared to the 2-phase protocol which requires extra state holding operators to detect the high or low state of a stable channel. The disadvantage of 4-phase signalling is that it requires twice as many transitions and hence is approximately twice as slow. Generally, 4-phase signalling is used on-chip where area costs are important and 2-phase signalling is used for inter-chip communication because of the large wire delays.

## 3.2 P/A-blocks

To use channels in high level processes, circuits are required which can manipulate channels. Two basic building blocks will be used, the P-block and the A-block. They will act as sequential operators, controlling the sequencing of channels. An example of their use is shown in Figure 3-1. The P-block corresponds to a passive input channel communication and the A-block corresponds to an active output channel communication. The process first takes in a passive channel signal from the P-block, and then sends out an active channel signal from the A-block. This sequence of events repeats itself because the process is circular as a result of the sequencing wire looping back to the top of the process. It can be seen that any combination of these P/A-blocks can be arranged to produce a sequential process consisting of passive and active channel events. The basic idea of these types of processes is the use of the

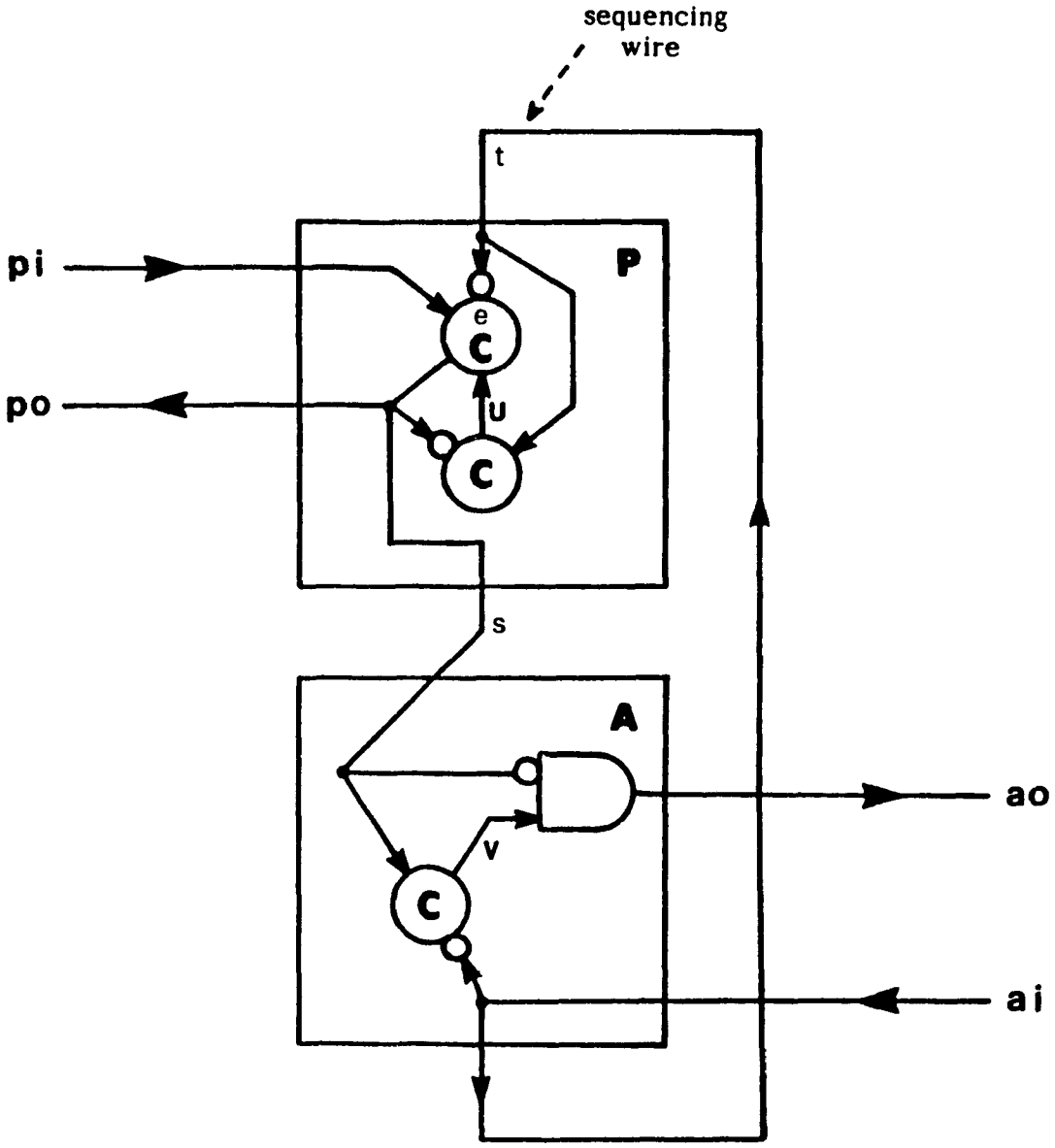



Figure 3-1: P/A-blocks.

sequencing wire to enforce the sequencing of channel signals. Blocks are connected together using these single wires to produce a sequential process. (Note that it is possible to optimize the P-block in special cases, such as when there is only one P-block at the beginning of the sequential process, but this form will be retained for the sake of generality.)

## Initial State:

At the initial stable state, all the channel wires are low. The state variable "u" in the P-block must be initialized to a one at system power on by presetting the output of the corresponding C-element. (A state variable is a storage node inside a module whose value can be a one or a zero, and a C-element is a gate whose output changes to the value of its inputs whenever all of its inputs are of the same value.) Since the inputs to this C-element is a one and a zero when the process is in the stable state, the output can take on any value. Since C-elements can act as storage elements they must be preset or cleared (just as in flip flops) whenever all of their inputs are not of the same value. This preset/clearing can be done using global lines at system power on, just as in synchronous systems. The state variable "v" of the A-block is also cleared to zero.

## Operation:

Since the first state variable is a one, a channel request signal is allowed to come in to the top P-block. Once the first input channel signal is completely finished (po goes low), the handshaking signals from the A-block will begin.

The A-block is activated from the incoming sequencing wire. The sequencing wire taps the second wire on the channel of the P-block. The result seen from the sequencing wire is a pulse because the acknowledge wire, po, first goes high and then low. Looking at the A-block, the sequencing wire s going high will set the state variable "v". When the wire signal goes low, the output channel handshaking signals of the A-block begin. The handshaking signals

therefore start on the negative edge of the sequencing wire. Since the negative edge signifies the end of the previous handshaking signals and the start of the next handshaking signals on the next channel, there is no mixing or shuffling of the signals between consecutive channels (this property of non-shuffled handshake signals vs. shuffled will be more clearly specified in the next chapter).

Once the channel signal for the A-block is finished, the negative edge will enable the P-block. The pulse from the sequencing wire t sets the state variable "u" when it goes high, the state variable "u" having been cleared during the previous P-block communication On the negative edge when the sequencing wire t goes low, the enabled C-element will become enabled and a channel request signal at pi is allowed to come in. (This enabled C-element may simply be replaced by an AND gate and an ordinary C-element, the output of the AND gate connected to one C-element input and the enable line being one input of the AND gate).

This procedure of sequencing from one channel event to the next occurs in a circular fashion, resulting in a continuously running process. If additional P or A-blocks are inserted, the sequential operation can be seen as a state variable rippling down from one block to the next. When a channel signal finishes, it clears the present state variable and sets the next.

## Delays:

Note that delays must be taken into account in these circuits. The C-elements should have the largest delays, compared to the gates, in order for the P/A-blocks to function correctly. Races between C-elements and gates must be handled properly. But once these local hazards are taken care of, the blocks can be used over and over again.

Another delay problem that may arise is the inter-block pulse on the sequencing wire which should be long enough to set the next state variable. These single wires are subject to transmission interference. This is not a problem

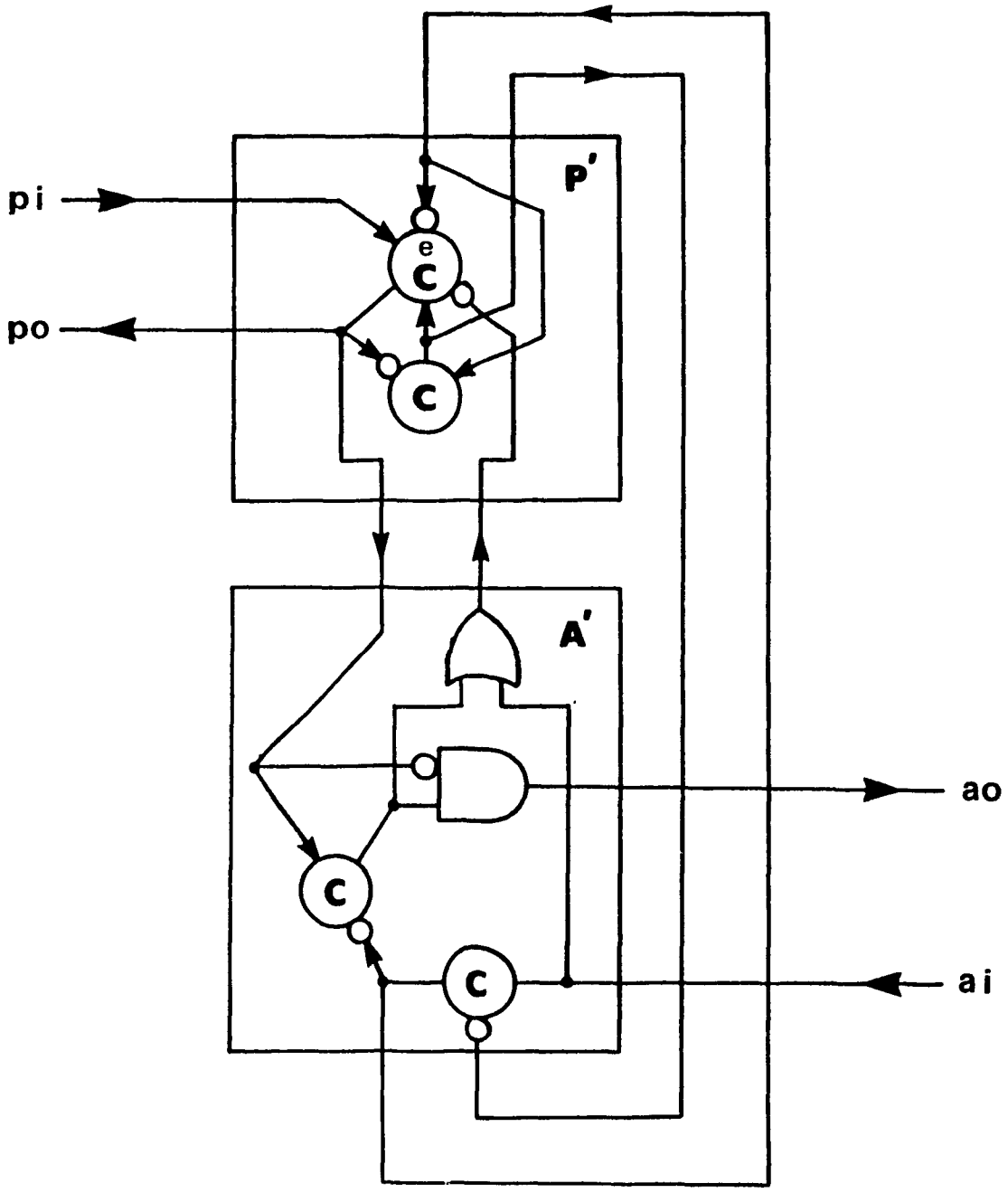



Figure 3-2: Channel P/A-blocks.

normally because the handshaking signal must pass through more gates than the number of gates used to set the state variable. The existence of this single wire means that these P/A-blocks are partially delay-sensitive. The composite sequential module, on the other hand, has solely channels at its interface and hence is delay-insensitive. The entire composite module is therefore considered as a DI module.

To avoid this inter-block delay sensitivity, channel P/A-blocks can be used as shown in Figure 3-2. The single sequencing wires are replaced by delay-insensitive channels. The interface of each block consists of channels only and hence these are finer-grain DI modules. There is no constraint on the wire lengths in the layout placement of each of these blocks. The drawback of course is the increased gate area required. In general, there is a tradeoff between finer grain delay-insensitive modules and the area required. The single wire P/A-blocks will be used in the next chapter as an example in the specification and composition of circuits which contain both delay-sensitive and delay-insensitive parts.

# Chapter 4

## Extended STG Model

The model that will be used to formally specify and analyze our circuits will be Signal Transition Graphs (STG's) [Chu87]. STG's are interpreted Petri nets, meaning that the places, transitions, and arcs of Petri nets are given certain semantics. As well, STG's are a subset of Petri nets because certain restrictions are made over general Petri nets. This model is chosen because it belongs to the class of models which are based on partial orders and so has the ability to clearly describe concurrent events. This is as opposed to another popular specification model for delay-insensitive circuits, trace theory, which is based on linear traces and hence does not describe concurrency very well.

In this chapter, a description of STG's is first given and then extensions to the basic model is described. These extensions allow STG's to describe circuits more clearly, distinguishing between events which are controlled by the environment and events which are controlled by the module. As well, delays are incorporated into the model. This allows for the specification of delay-sensitive parts of a circuit and therefore can clearly describe which paths of a concurrent circuit may need delay compensation and further analysis. The extended model can distinguish between delay-insensitive and non-delay-insensitive specifications.

A composition procedure is then described for hierarchically composing two STG specifications into a single composite STG specification. The composite specification describes the behavior of the resulting higher level module, and the lower level details are abstracted away. It will be shown that although two modules may be delay-sensitive by themselves, their composite specification can become delay-insensitive if proper delay compensation is done at the previous

level. Using such a composition procedure, the correctness of systems composed of individual modules can be verified.

## 4.1 STG Description and Extensions

STG's are a graph oriented model which are used for specifying asynchronous concurrent circuits. The original model was used as a specification tool for the synthesis of concurrent speed-independent control modules. The syntax of an STG is very similar to that of Petri nets. STG's are formally defined as $\Sigma_J = (T, R, M_0)$, where $T = J \times \{+, -\}$, $R \subseteq T \times T$, and $M_0 \subseteq R$. J represents the set of signals, T is the set of transitions of signal J, with "+" indicating a positive signal transition and "-" indicating a negative signal transition. R is the set of causality relations between signal transitions T and $M_0$ is the initial marking of the system.

An STG graph will consist of a set of vertices T, representing the plus and minus wire transitions of a circuit. Arcs (directed arrows) between the vertices are the relation R. These arcs represent ordering and concurrency of the signal transition vertices. An arc directed from transition $t_1$ to $t_2$ means that the occurrence of the signal transition $t_1$ will cause the occurrence of the signal transition $t_2$. Multiple arcs emanating from a single transition describes the concurrent occurrence of many signals. The initial marking $M_0$ describes the state that the system is initially in. The marking is indicated by round black tokens sitting on the arcs.

There is a direct correspondence between STG's and Petri nets. The signal transition nodes T of an STG correspond directly to Petri net transitions and the places of a Petri net are implied on the STG arcs. A token sitting on an STG arc is equivalent to a token sitting in a Petri net place. The firing sequence of the tokens in the two models is also identical. A transition is enabled to be fired when all of its input places (or STG arcs) contain a token. When the transition fires, a token from all of its input places is deleted and a token is placed on every one

of the transition's output places (or STG arcs). The firing of transitions is instantaneous, so signal transitions on a wire are modelled as occurring instantaneously.

In order to more clearly describe asynchronous circuits, the basic STG model will be extended. These extensions pertain particularly to delay-insensitive systems, providing the ability to clearly distinguish between delay-insensitive and delay-sensitive specifications. The notations used clarifies the relation between a DI module and its environment, specifying which events are controlled by the environment and which by the module.

The following informally describes the syntax and semantics of the extended STG model that will be used:

[a] : a is an input event to a module.

b : b is an output event caused by the module.

Ⓢ : s is an internal event inside a module.

[a] $\longrightarrow$ b : Input event a causes generation of output event b (controlled by the module).

b $\Longrightarrow$ [a] : Output event b causes the input event a (controlled by the environment).

i $\dashrightarrow$ j : Event i must precede event j, although i does not cause j (controlled by delays inside module).

i $\Rightarrow$ j : Event i must precede event j, although i does not cause j (controlled by delays from the environment).

a $\xrightarrow{t}$ b : Time delay t between occurrence of transitions a and b.

Double arrows are used to describe the events that are controlled by the module's environment, as opposed to single arrows which are controlled by the module itself. To describe delay-sensitivity, dashed arrows are used. These describe the ordering of events (signal transitions) that must occur for the correct

functionality of the system and is only controlled indirectly through the use of delays. The correctness of the system means that these events must be enforced through the use of delay analysis/compensation. This is as opposed to solid arrows in which the ordering between events is explicitly controlled by the logical connection of the circuit. Note that double dashed arrows are completely disallowed in DI circuits because we cannot control the delays produced by the environment of a DI module. Double dashed arrows must be converted into single dashed arrows by pulling this event inside of the DI module, after which delay compensation can be performed inside of the module.

A single solid arrow usually occurs when an input port affects an output port. The occurrence of an input transition will eventually lead to an output transition (if the module is live). Since the module is responsible for firing the output transition, this event is controlled by the module. Internal events may also be caused by an input transition and an internal event may cause an output transition. For double solid arrows, this occurs when an output transition of a module causes the environment to send in an input transition. This event is only controlled by the environment. Single dashed arrows can occur inside of a module, where we must have one transition precede another. These transitions do not directly affect one another in the logical connection of the circuit, but one transition must precede another to avoid hazards in the module. Double dashed arrows can occur when the environment sends a module two input transitions, one after another. If the two input transitions occur on a single wire, then the result is a pulse sent to the module. This event is controlled by delays in the environment, meaning that the environment controls the width of the pulse by varying the delay in firing the two consecutive input transitions. For correct operation, the module may require that the pulse be of a minimum width, such as for satisfying setup and hold times.

To clarify the different possibilities, we have

$$J = \{I, O\ N\}$$

where the signal J may be of three different types:

I = input transition
O = output transition
N = internal transition

The causality relation $R = \{L_M, L_E, D_M, D_E\}$ where

| | | | |
|---|---|---|---|
| L | = | direct logical relation | (solid arrow) |
| D | = | indirect delay-sensitive relation | (dashed arrow) |
| M | = | controlled by module | (single arrow) |
| E | = | controlled by environment | (double arrow) |

and the possible relations are

$L_M = \{I, N\} \times \{O, N\}$
$L_E = \{O\} \times \{I\}$
$D_M = \{I, N, O\} \times \{N, O\}$
$D_E = \{I, N\} \times \{I\}$

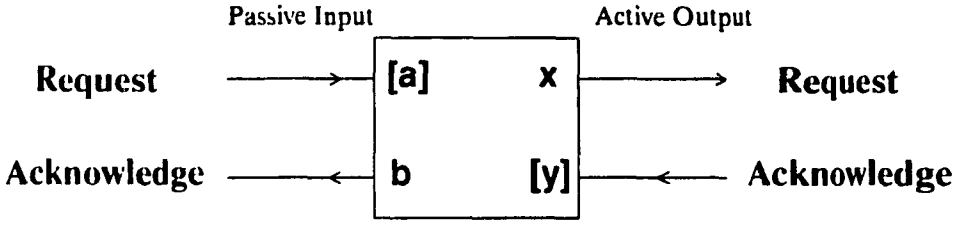
A delay-insensitive specification requires that $R = \{L_M, L_E\}$ only. All relations must be direct logical causality. A delay-insensitive STG specification of a DI module must consist of only solid arrows with no delay-sensitive dashed arrows. In addition, it is required that $L_M = \{I\} \times \{O\}$. A specification of a DI module requires that it does not contain any internal event nodes N. Otherwise, if the STG contains internal nodes, then it is a white box implementation rather than a black box specification. (A black box means that the internals of a module cannot be seen; only the I/O ports can be seen. A white box "sheds light" onto the module, and we can see inside the module.) Although it is required that a DI specification be free of both delay-sensitive dashed arrows and internal nodes N, it is possible to have an STG contain internal nodes and still be delay-insensitive. Such a case occurs when we want to describe lower level implementations which are still DI and have not yet reached the lowest level delay-sensitive parts.

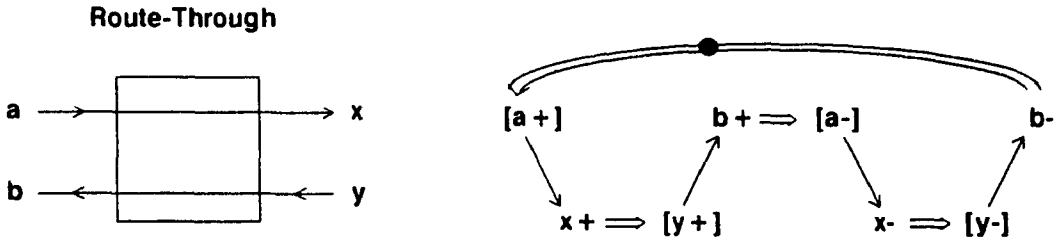
## 4.2 Example of Delay-Insensitive STG Specifications

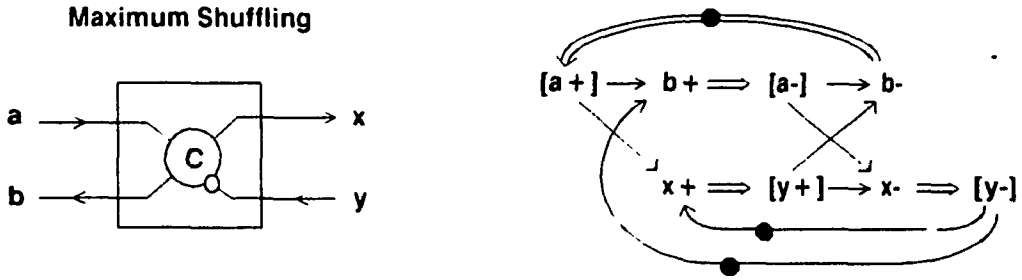
As an example of STG specifications of DI modules, Figure 4-1 shows some possible specifications of the basic four wire module. The module consists of two channels, one passive input channel and one active output channel. The input and output port of the passive channel are [a] and b, respectively, and [y] and x for the active channel. Such a module in which an input channel signal causes the occurrence of an output channel signal means that the two channeis will interact with each other. The individual request and acknowledge wire transitions can occur at different times, either sequentially or concurrently. This results in a shuffling of the handshake signals.

For example, when the handshake signals of the two channels are not shuffled, the trace is ([a+]; b+; [a-]; b-; x+; [y+]; x-; [y-])*, where the asterisk denotes repetition. The entire set of handshake signals of the left channel is completed, followed by the set of handshake signals for the right channel. The handshake signals between the two channels are not intermixed. If shuffling is done, such as the example for maximum shuffling, the trace is ([a+]; b+; x+; [y+]; [a-]; b-; x-; [y-])*. It is called maximum shuffling because all the positive transitions are moved to the left as far as possible and all the negative transitions are moved to the right. If there were an additional channel, then its positive transitions would also be moved to the left. The requirement for shuffling is that the cyclic order of the four-phase signal transitions of each individual channel be preserved.

The disadvantage of traces is that it cannot adequately describe concurrency. For example, the above trace for maximum shuffling does not describe the concurrency between the transitions [y+] and [a-]. The STG specification explicitly describes this, as well as any other dependencies. The reason is that in the above trace, the semi-colon does not correctly describe sequencing; it only loosely describes ordering. The situation can partially be alleviated by introducing a comma notation to describe concurrent signals, but



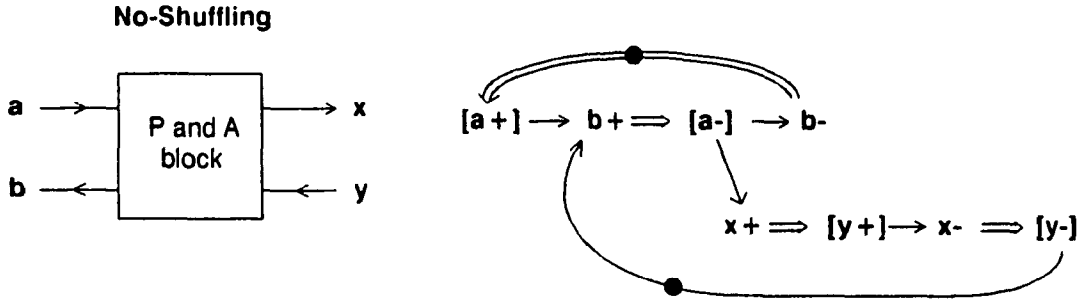




**Figure 4-1: STG examples of the basic four wire module.**

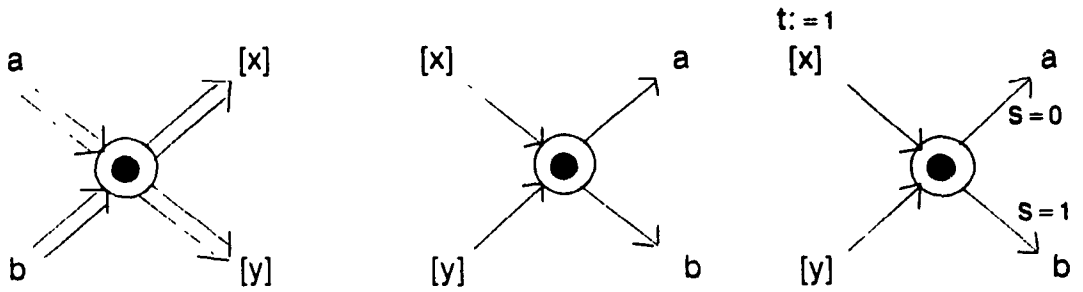
will not be adequate to describe every form of concurrency at different process levels.

The three examples of Figure 4-1 show non-shuffled, maximum shuffling, and route-through types of channels. Note that their STG specifications contain no delay-sensitive dashed arrows, hence the module is delay-insensitive for all three cases. This thesis will concentrate on only the two types of non-shuffled and route-through channels in the implementation of DI circuits and systems. The non-shuffled type of circuit has already been described previously in the P/A-blocks. The STG for no-shuffling in Figure 4-1 is the specification of the P/A-blocks for the special case in which one P-block is connected to one A-block. This is also the specification for a non-shuffled quick-return FIFO element. The route-through channel example in Figure 4-1 consists simply of two wires. It is called route-through because a channel signal coming in is directly routed through the module. In more complex modules, there will exist more than one channel interface to the module which uses route-through channels.

## 4.3 Choice and Non-Determinism

The description of non-trivial circuits using STG's require the ability to specify choice and non-determinism in addition to the fixed deterministic events. Circuits such as FIFO elements and control modules are deterministic cyclic processes whose signal transitions always repeat in the same order. More complex processes require the flexibility of allowing different data values to be input to the process. A major issue in the area of concurrency concerns the use of arbitration. In specifying choice, it can be either deterministic or non-deterministic; in addition, a choice can be either controlled by the environment or by the module. There are three possible types of choices (from the point of view of the module):

(a) non-deterministic choice from the environment
(b) non-deterministic arbitration in the module
(c) deterministic choice from state variables in the module




(a) External choice (non-deterministic environment)

(b) Internal choice (non-deterministic arbitration)

(c) Internal choice (deterministic state variable)

**Figure 4-2: Choice in STG's.**

The specification of these three types of choices using STG's is shown in Figure 4-2. It requires the explicit use of places as in Petri nets, rather than having all the places implied on the STG arcs. Choice occurs similarly to that of Petri nets. There exists an input place which is shared by two or more transitions, with only one token sitting in the place. When one of the transitions fire, it removes the token and prevents the other transitions from firing.

In the diagram, it can clearly be seen which choice is controlled by the environment and which by the module. If the place sits on double arrows, then the choice is made by the environment; if the place sits on single arrows, then the choice is controlled by the module. Choice from the environment is the selection of which input signal transitions to fire, such as in the input of different data values. This type of choice is called non-deterministic (from the point of view of the module) because the selection of different values will result in different actions in the module; the module does not always follow the same course of action. This type of environment choice is also called "input choice" by Chu in the original STG model. But the model does not describe non-deterministic choice controlled by the module, as shown in Figure 4-2(b). This type of internal

choice is also known as arbitration. This extension is required for the correct specification of circuits containing arbiters and mutual exclusion elements.

The third type of choice (called "non-input" choice by Chu) occurs when state variables are used in the module. State variables are storage elements which record the present state of the module. Depending on the present state, different courses of action may occur. As shown in Figure 4-2(c), a token sitting on a place may follow two different paths. If the present state variable s = 0, then transition a fires, else if s = 1, then transition b fires. The circuit implementation of this will usually imply an asynchronous flip-flop type of element controlling a multiplexer or demultiplexer type of circuit. The present state can change when another transition in the graph fires. The example shows the input transition [x] associated with the assignment of the state t: = 1 (assignment ": =" and equivalence " = " are borrowed from Pascal). When this transition fires, the state variable t becomes a one. In a delay insensitive specification, only input transitions are allowed assignment statements.

Note that the s = 0 and s = 1 cannot both be set at the same time; otherwise, it will not be deterministic. For verification of determinism, all the transitions in the graph that make assignments to the state variable can be checked if they can ever fire concurrently. If any two of these transitions may fire concurrently, then there may be non-determinism. The circuit equivalent of this situation is when we try to set and reset a flip-flop at the same time, resulting in non-deterministic metastability. As well, because of different delay paths, the order in which the state variable changes can affect the operation of the circuit.

This type of behavioral notation can be simulated structurally with a Petri net by adding additional places and transitions. We use this notation for the sake of decreasing the complexity of the graph in specifying circuits. Since the basic STG model is now extended to allow explicit places (and later to multiple tokens in places), we mainly use STG's rather than Petri nets because of the conciseness of notation and the clearer behavioral description of DI modules.

Note that a distinction is made on the types of internal memory that a module may have. There can be two types of memory:

1) memory from pipelining
2) memory from state variables

Pipelined memory occurs in circuits such as FIFO queues. The FIFO queue may be empty or partially full. The STG for such a queue (and in general, pipelined circuits) is specified by the presence of an increased number of tokens in the graph and their particular location in the graph. A queue that is partially full is specified by the difference in location of its tokens between its present state and its initial state of being an empty queue. At the circuit level, filled queues are characterized by pending requests. As will be seen in the section on composition of STG's, the specification of a FIFO queue with a larger buffer size will contain more tokens than a queue with a smaller buffer size.

Memory from state variables, on the other hand, will be used for internal choice. Its specification is the notation described in this section. (Note that the state variables in the P/A-blocks are used for controlling the handshake sequence of channels. They are deterministic and free of choice. They only occur at the lowest level building blocks for controlling the handshake sequence and are not considered as memory here). Therefore, the global state of a module will consist of both (i) the state of all the internal-choice state variables in the STG graph and (ii) the position of the tokens in the graph.

## 4.4 Hierarchical Composition of STG Specifications

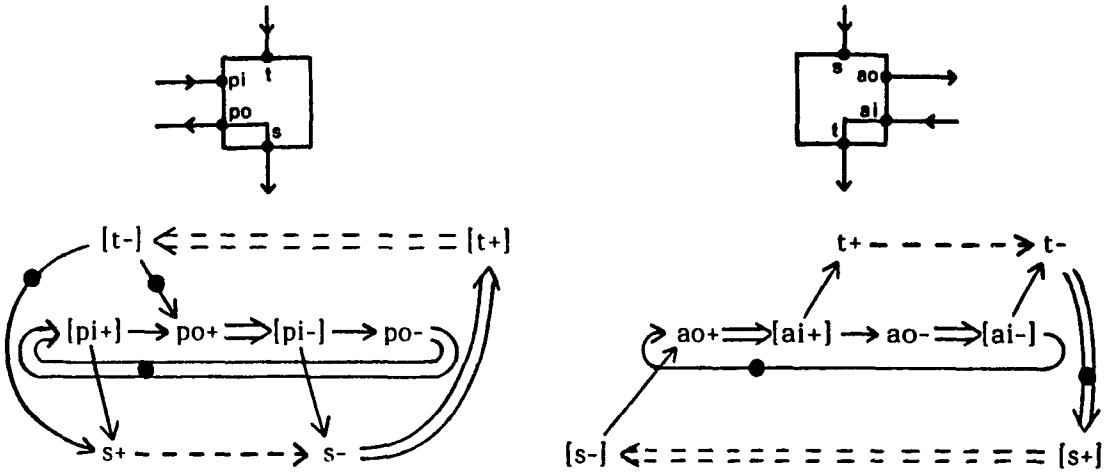
Composition is an important method used for the verification of systems. To verify the correctness of a system consisting of a network of modules, the specified behavior of each individual module in the network must be composed together to form a composite behavior of the system. The resulting composition is then compared with the desired behavior of the original system. In order to be able to make the comparison, the behavior must be a unique specification. As there are many possible implementations for a desired behavior, it is required

that specifications rather than implementations be composed and that the result be a unique specification. This separation between specification and implementation allows one to create modules using any desired method, as long as the specification remains the same. Composition can then be done without regard to implementation details.

In order to manage complexity, it is required that a composition technique be hierarchical. At each level of composition, the resulting specification for the composite network of modules should no longer contain unnecessary details of the network. The abstracted composite module should only retain essential information describing the behavior of the network. Memory and states which control the ordering in which I/O ports fire must be retained in the resulting specification, while unnecessary information such as internal nodes should be deleted. In this way, it is ensured that the resulting specification be unique so that it can be further hierarchically composed (if it is not unique, then the result is an implementation and not a specification). For example, the specifications of each module in the network may contain a different number of state variables. Once composed, the resulting module should contain a composite state information, without regard to the possibility of redundant states in the network. Internal nodes and wire connections of the network are no longer valid, as there may be many different networks that satisfy the same composite specification. Each level up the hierarchy, the network connection of the individually specified modules become an implementation and the composite module becomes the new higher level specification.

As an example of how modules specified with the extended STG model are composed, Figure 4-3 shows the composition of the P/A-blocks. These blocks are chosen to demonstrate how delay-sensitive components can be composed to form a delay-insensitive module. Although each individual block is sensitive to delays, the new higher level composition is delay-insensitive if compensation is done at the previous level. Once we obtain the composite DI module, the

**Step #1:** Specify modules to be composed as extended STG's.




**Step #2:** Compose modules by connecting the desired wires together. This corresponds to drawing an arrow from output transitions to input transitions.

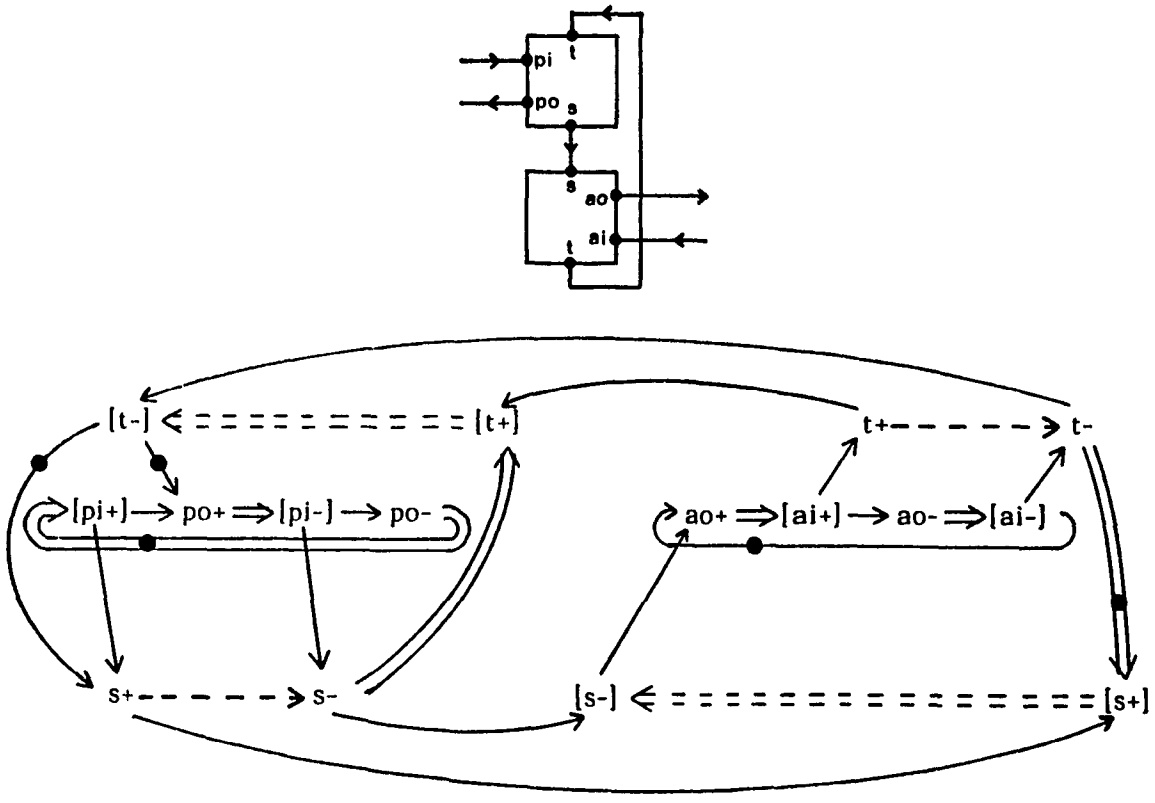
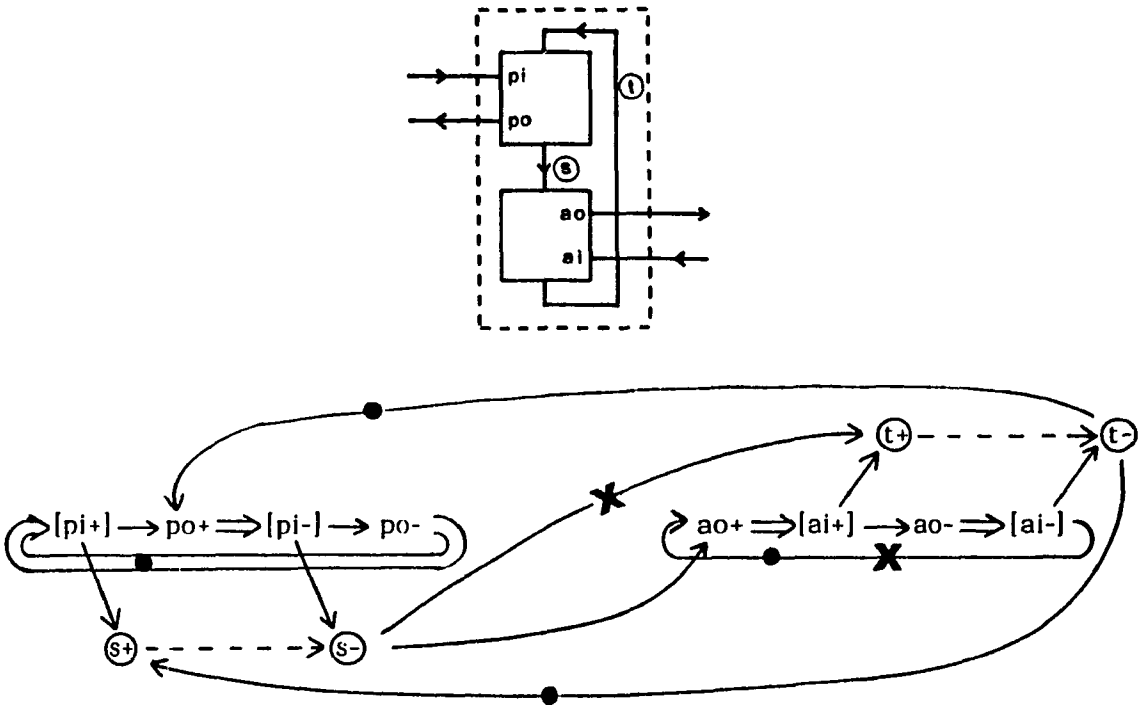



**Figure 4-3: Composition of the P/A-blocks.**

**Step #3:** Transform the composition into a single module. This means
1) collapsing connected nodes into single internal nodes,
2) converting double arrows into single arrows if they become internal to the module.




**Step #4:** Delete unnecessary arrows (arc deletion).

**Step #5:** Delete internal nodes (node deletion).

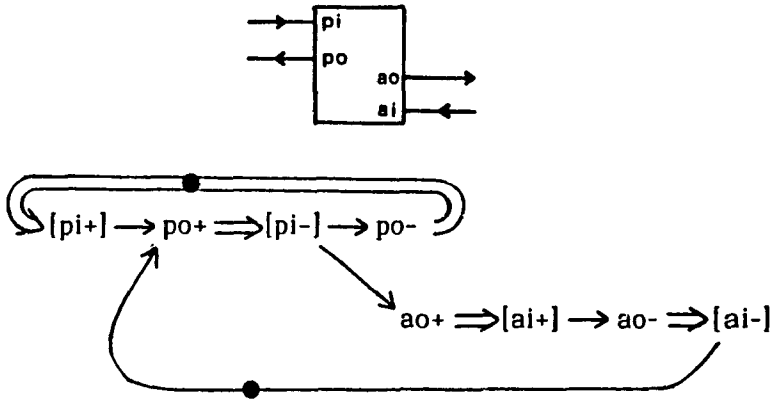



**Figure 4-3: Composition of the P/A-blocks (continued).**

delay-insensitive specification can be composed further to form another delay-insensitive specification.

In Figure 4-3, the steps shown to compose two STG specifications have a direct correspondence to connecting two circuit modules together. In the first step, the individual modules are specified. Note that only the I/O ports of the modules appear in the specification; internal state variables controlling the 4-phase handshake sequence should not be shown. In step #2, the modules are connected together via their single sequencing wires. This corresponds to a direct weave of the two STG's, drawing an arrow from the output transitions to input transitions. In step #3, the connected blocks are transformed into a single module. The internal wire connections become single variables and the I/O ports of the individual blocks become the I/O ports of the composite module. In the STG, the previously connected nodes are collapsed into single internal nodes (transitions). Double arrows are converted into single arrows if they become internal to the module, as events that were previously controlled by the environment now become internal causality. In step #4, arc deletion is done (arrows marked with an "X"). Arcs which are redundant and do not affect the behavior of the system are deleted. At this point, we have the behavior of the system that also includes the internal nodes. To get the unique specification of the composite module, these internal nodes must be eliminated by node deletion.

Note that since STG's are interpreted Petri nets, arc deletion corresponds directly to the deletion of Petri net places and node deletion corresponds directly to the deletion of Petri net transitions. (An algorithm for deleting Petri net transitions called "net contraction" appears in [Chu87b], page 99. It is used in the decomposition of finite automata, but can also be applied here.)

The final resulting module is the STG specification of a non-shuffled quick-return FIFO element. It is delay-insensitive because it does not contain any dashed arrows which reflect delay-sensitivity. The previous STG, on the other hand, does contain dashed arrows. This demonstrates the cut-off point with which delay-sensitive circuits become DI. For correct operation of the higher

level module, this delay-sensitivity must be analyzed at the lower level, and if necessary, delay-compensation done (this will be done in the next section). The higher level module can then be used in totally delay-insensitive composition.

As an example of totally delay-insensitive composition, the non-shuffled FIFO element, obtained from the composition of the P and A-blocks, can further be composed. Figure 4-4 shows the composition of two of these single buffer FIFO elements to form a FIFO element with a buffer size of two. The composition procedure is similar to before. The specification of the single buffer non-shuffled FIFO element is first obtained. A direct weave is then done on the two specifications, drawing a directed arc from the output of one module specification to the input of the other module specification. The pairs of these connected output-to-input transitions are then collapsed into single internal nodes, described in this example by the variables a and b. To collapse the pairs of transitions into internal nodes, arc deletion can first be done on the graph to delete the unnecessary arcs. Then to obtain the single collapsed node, either the input or the output of each transition pair may be deleted using the net contraction algorithm. Once we have the graph specification that contains the single internal nodes (this graph is a delay-insensitive implementation), the STG specification of the composed system is obtained by deleting the internal nodes. Deleting the internal nodes can be done again by the net contraction algorithm of Chu. Each of the internal nodes are deleted one by one. Essentially, given x $R_1$ y $R_2$ z where x, y, and z are transitions and $R_1$ and $R_2$ are the dependency relations, the internal node y is deleted by drawing an arc from x to z; the number of tokens on the new arc is obtained by summing the tokens on $R_1$ and $R_2$.

From this example, one can see that arc deletion is not necessary in the actual composition procedure. The extraneous arcs does not affect the net contraction algorithm which is used first to obtain the single internal nodes and then to delete the internal nodes Doing arc deletion does not affect the firing behavior of the graph. Arc deletion should be done at the beginning and at the end of the composition procedure where we want unique specifications. More

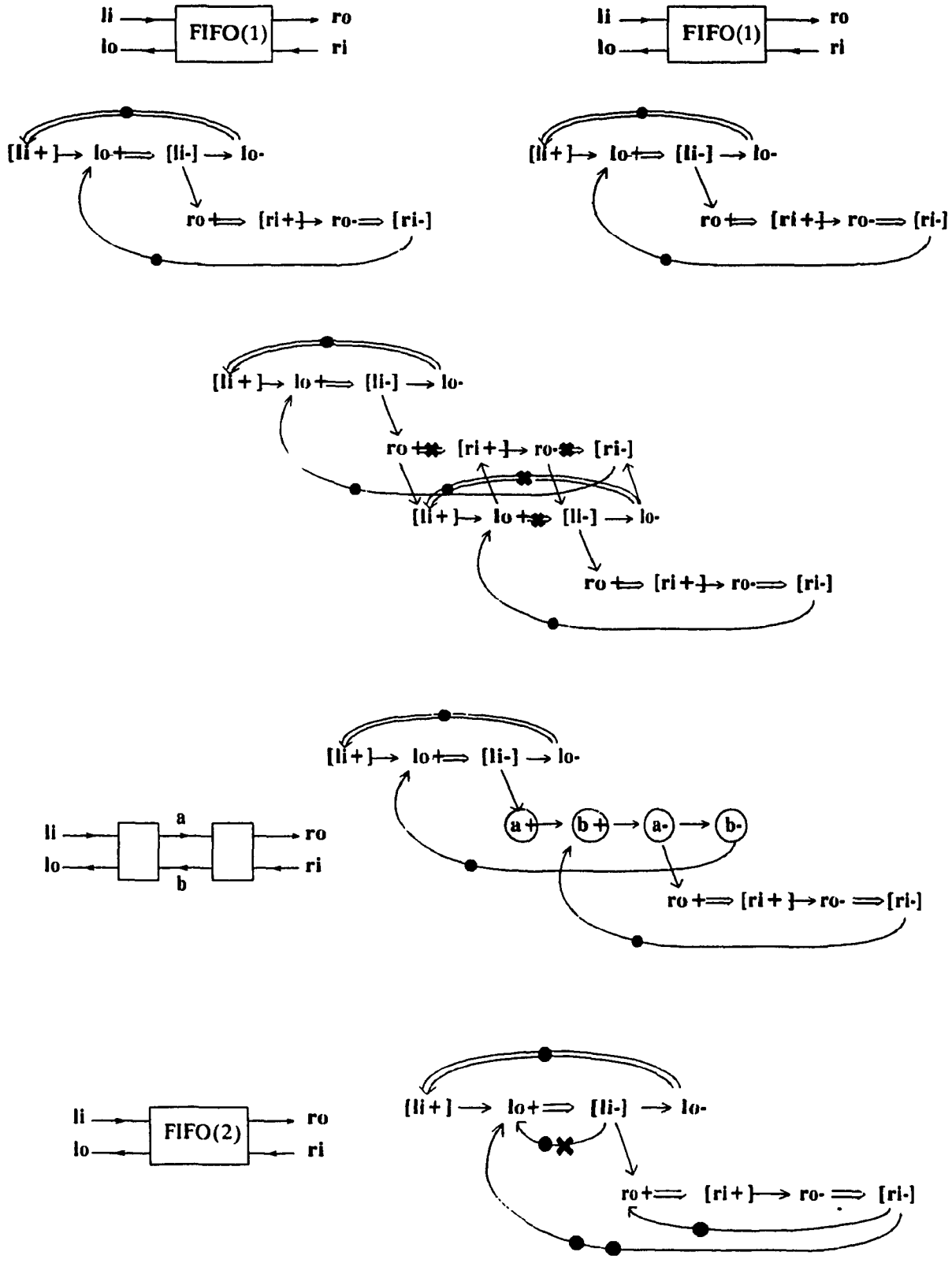


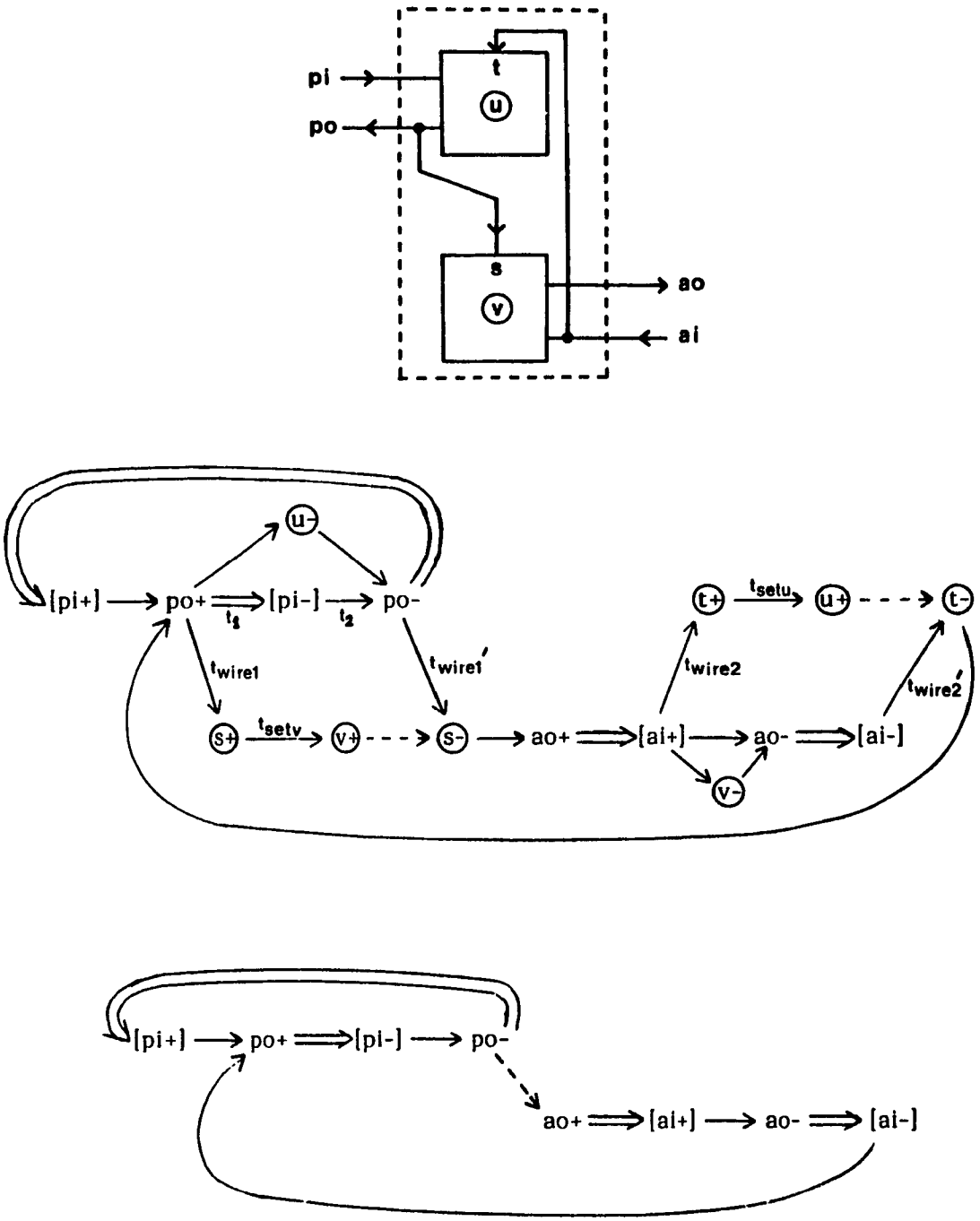

**Figure 4-4: Composition of non-shuffled FIFO elements.**

work is required in writing the algorithm for arc deletion (deletion of Petri net places), although it does not appear very difficult. The complete general algorithm for the composition of extended STG's is left for future work. The algorithm should take into account non-deterministic and deterministic choice, as well as higher level channel signal STG's and data tokens.

## 4.5 Delay Analysis

In the previous section, the composition of the single sequencing wire P/A-blocks was sensitive to delays as indicated by the dashed arrows. In order that the resulting composed FIFO element be delay-insensitive, it must be ensured that the delay constraints are satisfied. This section shows how such a delay analysis is done using the extended STG model. (Note that the delay-sensitive single wire P/A-blocks are used only as an example of how delay analysis can be done and by no means advocates its use over that of the totally delay-insensitive channel P/A-blocks. It is up to the user to decide whether the use of delay-sensitive building blocks is desired for area optimization, at the cost of some delay analysis.)

To precisely pinpoint where the delay-sensitivity arises and how compensation can be done, the lower level STG of the white box implementation is shown in Figure 4-5. The STG includes the internal state variable nodes u and v which control the 4-phase handshake sequencing. It also includes the labelled delay times between transitions. The lower level STG reveals two delay-sensitive areas, as indicated by the dashed arrows. In order for correct operation, the transition v+ must precede s- and the transition u+ must precede t-. The state variables u and v must set before the negative transitions s and t occur. (The resetting of the state variables, on the other hand, is not sensitive to delays because u- is a direct logical causality of po- and v- a causality of ao-.)




(The output affecting another output is caused by the actual implementation; the result still follows the FIFO element specification)

**Figure 4-5: STG of the lower level P/A-block implementation.**

Since the dashed arrow precedence relationship is not directly controlled by the logical connection of the gates, the ordering must be controlled by the proper compensation of delay paths. It is required that the following inequality be satisfied for the dashed arrow from v+ to s-:

$$t_1 + t_2 + t_{wire1}' > t_{wire1} + t_{setv}$$

where $t_1$ = time the environment takes to send the module the negative acknowledgement signal

$t_2$ = delay of the P-block

$t_{wire1}$ = delay for positive transition to travel along sequencing wire to next block

$t_{wire1}'$ = delay for negative transition to travel along sequencing wire to next block

$t_{setv}$ = delay time for setting the variable v

and similarly for the dashed arrow from u+ to t-. As long as this inequality is satisfied, the state variable v will set before the negative transition of s. (For simplicity, the path through the internal node u- is not considered.)

If it is found that the inequality does not hold, then delay compensation is done through the insertion of a delay element to increase the left hand side of the inequality. This corresponds to increasing the upper delay path in the graph. The delay element should be placed so as to increase the delay $t_2$, which means putting the delay element inside the P-block module. Inserting the delay element on the sequencing wire increases the time $t_{wire1}'$, but will be cancelled by a corresponding increase in time $t_{wire1}$. The delay $t_1$ is controlled by the environment, so compensation in the composite DI module cannot be done to increase this delay. Theoretically speaking, a DI module should make no assumptions about the external delays of the environment, in which case $t_1$ should be equal to zero in the worst case. But practically speaking, it can be guaranteed that the channel will have a minimum delay at the other end, such as being terminated by an A-block, the source of the channel signal. In such a case, a lower bound can be placed on the delay $t_1$. By placing lower bounds on the delays of the environment, the possibility of requiring delay elements for

compensation is reduced. For this P/A-block example, if the sequencing wire is short and has a negligible delay (an isochronic wire assumption is made), then there is probably no need for an extra delay element.

# Chapter 5

## Modelling 4-Phase Handshaking with Packets

Packets are introduced as a method for modelling the 4-phase handshake communication protocol. Handshake signals are a requirement for delay-insensitive systems, which may use either explicit or implicit handshaking, 4-phase or 2-phase implemented with two wires, or a derivative involving more than two wires. The packet model presented in this chapter is oriented towards the simplest type of channel, that of two wires operating on the 4-phase handshake protocol. In particular, the channel is of the non-shuffled quick-return type. Extensions can be made to incorporate more complex channels.

The primary purpose of using packets is to provide an abstraction for the detailed lower level 4-phase handshake communication signals. The simplification results from being able to interpret the four separate wire transitions as being only a single packet travelling on a channel. The idea of packets is borrowed from data communications in which asynchronous data messages are transmitted between two sites. But instead of using packets for only high level communication, they can be used in the lower level circuits. The implication is that VLSI circuits can be designed at the packet level, rather than having to deal with the complexities of lower level gates and wires.

The modelling of 4-phase handshake signals with packets is limited to the non-shuffled quick-return type of circuits (for reasons to be explained later). 4-Phase handshaking and packets have three properties in common:

1) direction
2) duration
3) discreteness

The first property of direction is obvious. Any handshaking signal must be terminated by an active and a passive end of a process module. There must be

an active end to initiate the request and a passive end to give the acknowledgement. A packet travelling on a channel also has direction, going from one module to another.

The second property is duration. There is always a certain duration required to complete a 4-phase handshaking signal. The duration comes from the switching delays of the gates and the time for a signal to travel over a wire. A high signal is sent and received by the active process module on the first half of the cycle, and a low signal is sent and received on the second half of the cycle. A packet can also have a delay in travelling from one module to another. The duration will be the in-flight time of the packet while it is travelling through a channel. The creation and sending of a packet is equivalent to a request signal being made, the 4-phase handshaking signals being in progress is equivalent to a packet in flight, and the end of the 4-phase handshaking signal is equivalent to a packet being absorbed by a process at the receiving end.

The third property of discreteness applies to the non-shuffled, quick-return type of handshaking. A single packet or handshaking signal sent on a channel is discrete; only one packet can exist on a channel at a time. There is no overrunning of packets from one channel to the next because acknowledge signals must be received before the next request is sent. In non-shuffled circuits, a set of handshaking signals at the input of a process must be completely finished before they cause the next set of handshaking signals at the output of the process to start. *This means that a single packet is completely absorbed on an input channel before any output packet induced by this input packet is sent to an output channel.* This is not true for shuffled handshaking signals; the sending of a single packet may result in more than one channel being active, and hence, more than one packet being produced as a result of only a single packet signal.

To clearly illustrate the property of discreteness for non-shuffled, quick-return type of circuits, an example is shown in Figure 5-1. Three discrete packets are shown modelling the three active channels with their 4-phase handshake signalling in progress Packet #2 sent out from process #1 will make

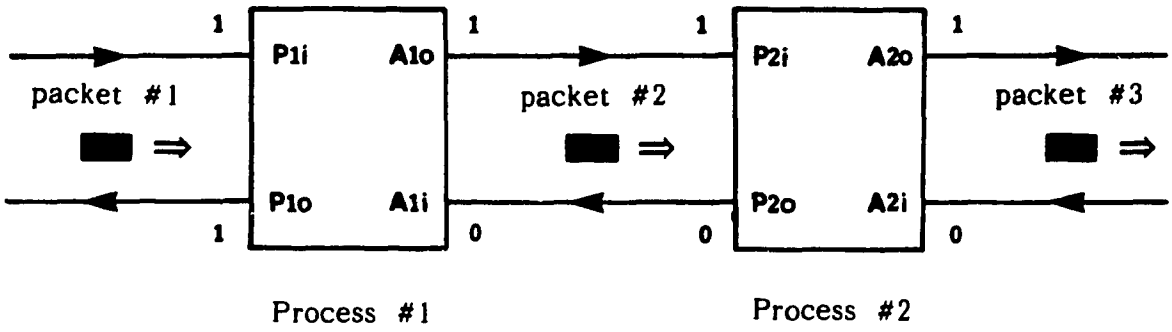



**Figure 5-1: Example of discrete packets for modelling 4-phase handshaking in non-shuffled circuits.**

the output channel of the process active (A1o = 1). Packet #2 is discrete in the sense that a packet signal at the input channel (P1i = 1) is not the cause of that output packet. Furthermore, the output channel of process #2 being active (A2o = 1) is not caused by the packet #2 being active at its input channel.

This is not the case for the shuffled types of circuits where they do not wait for the full handshake signal at the input to be completed before sending the output signal. For example, P1i going high can cause both P1o and A1o to go high. A1o will in turn cause P2i and A2o to go high as well. The single original packet, therefore, can cause more than one channel to become active, and so discrete packets will not have a one-to-one relationship with the channels.

Semantically speaking, a single handshaking sequence will always be discrete in the sense that this signal is always treated as a single signal as it travels throughout the circuit. But in the case of shuffled circuits, this one packet gives rise to many active channels, and it will be difficult to visualize the handshaking signals using a packet model. Non-shuffled circuits, on the other

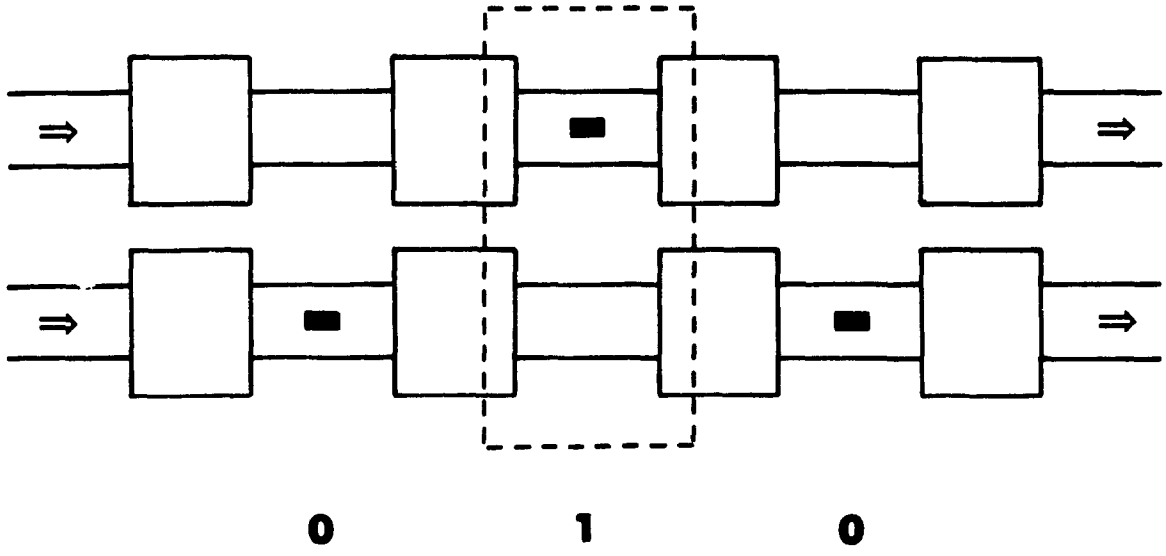



**Figure 5-2: Non-shuffled full FIFO queue modelled with packets.**

hand, separate the handshaking signals as much as possible and is easily modelled by discrete packets.

The intention of the packet model is for it to be used in high level system design. Since packets do not accurately describe shuffled circuits, a correct composition at the packet level may lead to deadlock if shuffled circuits are involved.

An example of using packets to better visualize handshaking signals is in the non-shuffled full FIFO queue, as shown in Figure 5-2. The queue consists of pairs of quick-return FIFO elements, each pair enforcing mutual exclusion so that a signal can represent either a one or a zero. A single packet can enter from either the top or bottom channel and flow through the queue. The queue may be filled up by not giving acknowledge signals to the queue at the extreme right. Since the handshaking signals are stuck in the middle of the sequence, the packets are also stuck in flight in the middle of the channel.

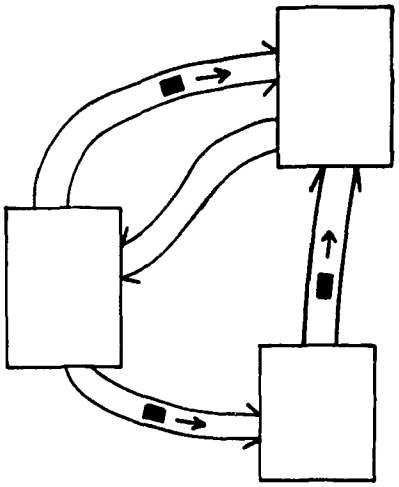


**Figure 5-3: Packet communication in high level processes.**

An extension to the packet model is the use of bit packets. Bit packets travel on pairs of mutually exclusive channels that represent one binary bit. The existence of a single packet on either the top or bottom channel represents a bit packet with a value of one or zero. Bit packets can be colored as white for a one and black for a zero.

Using packets (or alternatively tokens) to model the handshaking signals greatly eases the visualization of how the circuit operates. Design complexity is reduced when lower level details are abstracted away. Looking at the individual transitions of wires can become confusing, especially when higher level processes are constructed.

An example of higher level processes is shown in Figure 5-3. Each process communicates via quick-return channels, so communication can be visualized as the flow of packet signals from one process to another. Each process is a DI module. The channels are drawn using double arrows as shown, and the existence of packets in the channels indicate a particular state that the system is in.

# Chapter 6

## DI Building Blocks

To construct delay-insensitive systems, a set of basic building blocks is described. These DI building blocks are meant to be the lowest level constructs with which delay-insensitive systems are to be built. They are used at the bottom level of the hierarchy. As the building blocks are composed together, they will progressively form higher level DI modules. The building blocks are referred to as fine-grained because they are most nearly the smallest size DI modules possible. Large-grain DI modules, on the other hand, would require that the internal delays of a larger area be taken into account. The advantage of using fine-grain DI modules is that there is better layout freedom of the circuit. Design automation is facilitated when there is freedom in placing and routing finer grain DI modules which are limited to a basic set. Of course, the user is free to substitute specially designed DI modules to optimize the speed and area, but at the cost of higher design time and complexity.

This chapter will first give a general description of the DI building blocks and characterize them using the packet model. The blocks are then formally specified using STG's. Circuit implementations are given and then extensions of the basic building blocks are shown.

### 6.1 General Description and Characterization

To synthesize a delay-insensitive system, there must be a basic set of operators which are powerful enough to describe any general circuit that is desired. The operators must be able to describe concurrency, memory, and choice, as well as control operations. In the area of concurrency, provisions must be made for arbitration. There are different possibilities for the selection of operators. For example, the operators can be the basic AND, OR, inverter, and

wire. Alternately, they may even be transistors at a lower level. At a higher level, the basic operators can be CPU's, registers, and RAM modules. In the area of self-timed systems, the basic operators are most commonly individual gates, supplemented with C-elements for the synchronization of asynchronous signals. The construction of speed-independent circuits also commonly uses gates as the basic operators. But in delay-insensitive systems, the use of gates as the basic operators for synthesis is difficult, if not impossible. This is because gates are inherently delay-sensitive. A strategy is required to make the jump from delay-sensitive gates to the delay-insensitive DI module level. In speed-independent circuits, gates are sufficient operators because of the simplifying assumption of isochronic forks and wires. Delay-insensitivity on the other hand must satisfy the two sided requirement of insensitivity to the delays of both the operators and the wires. It is difficult to foresee the synthesis of strictly delay-insensitive systems with gates as operators without having to resort to making some "negligible wire delay" assumption at the lowest level (in which case it will become speed-independent and not delay-insensitive). Speed-independent techniques may be used to construct DI modules, after which the DI modules can be used in delay-insensitive composition.

The basic operators with which we will build delay-insensitive systems are called DI building blocks They are shown in Figure 6-1. They are the next step up from the gate level. Each of the building blocks are DI modules because they communicate using solely delay-insensitive channels (a channel is shown as a double arrow to indicate a request-acknowledge wire pair). The interface of each module is DI, while the internal implementation may be sensitive to delays. These building blocks are said to be fine-grained because the modules are nearly as small as they can be and still maintain a DI interface. The implementation of these modules may take on any form, either constructed out of gates or optimized at the transistor level, be speed-independent or use delay compensation. These DI building blocks are intended to be sufficient to build any delay-insensitive system. The route-through constructs are used for the manipulation of channels.

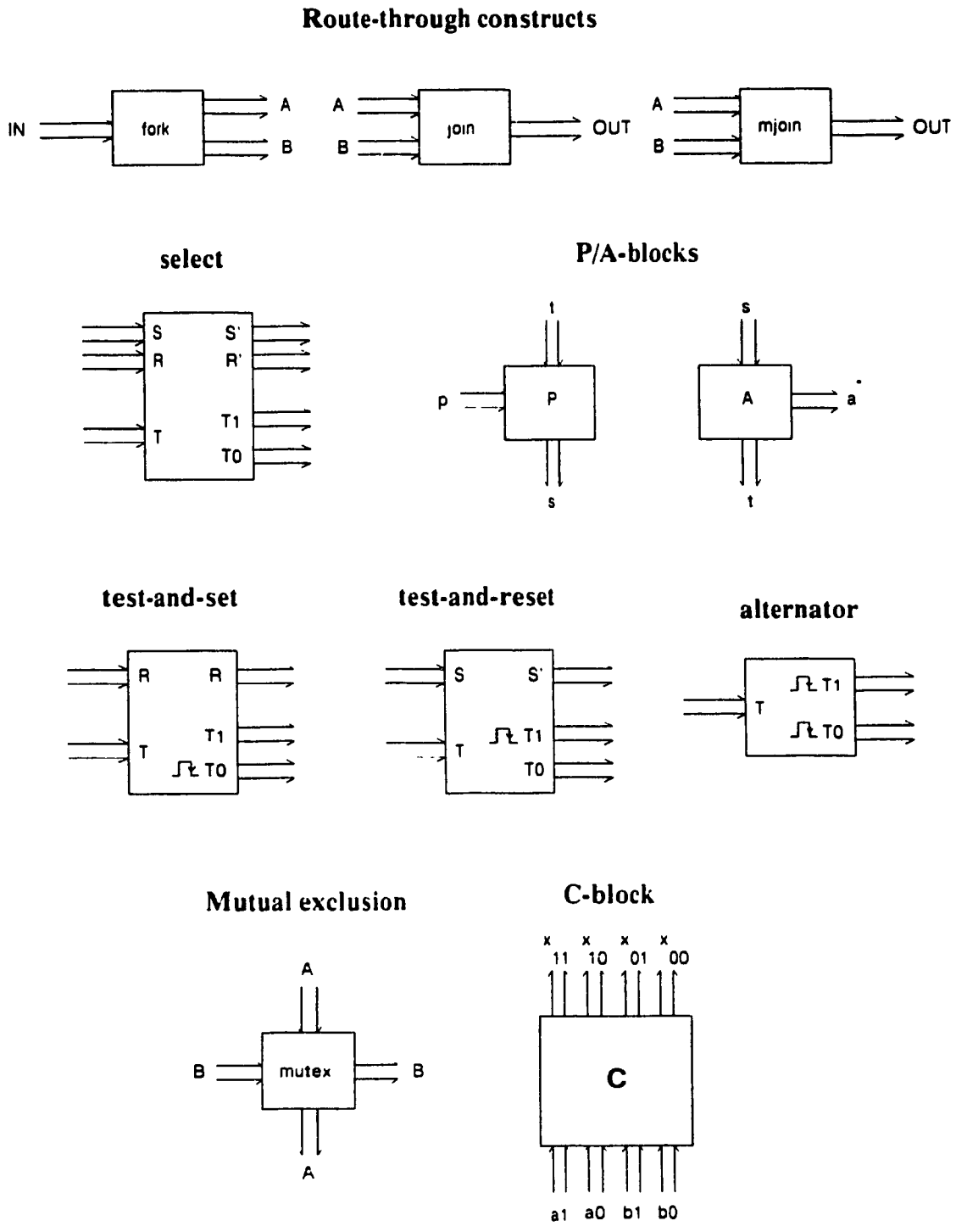



**Figure 6-1: Basic set of DI buiiding blocks.**

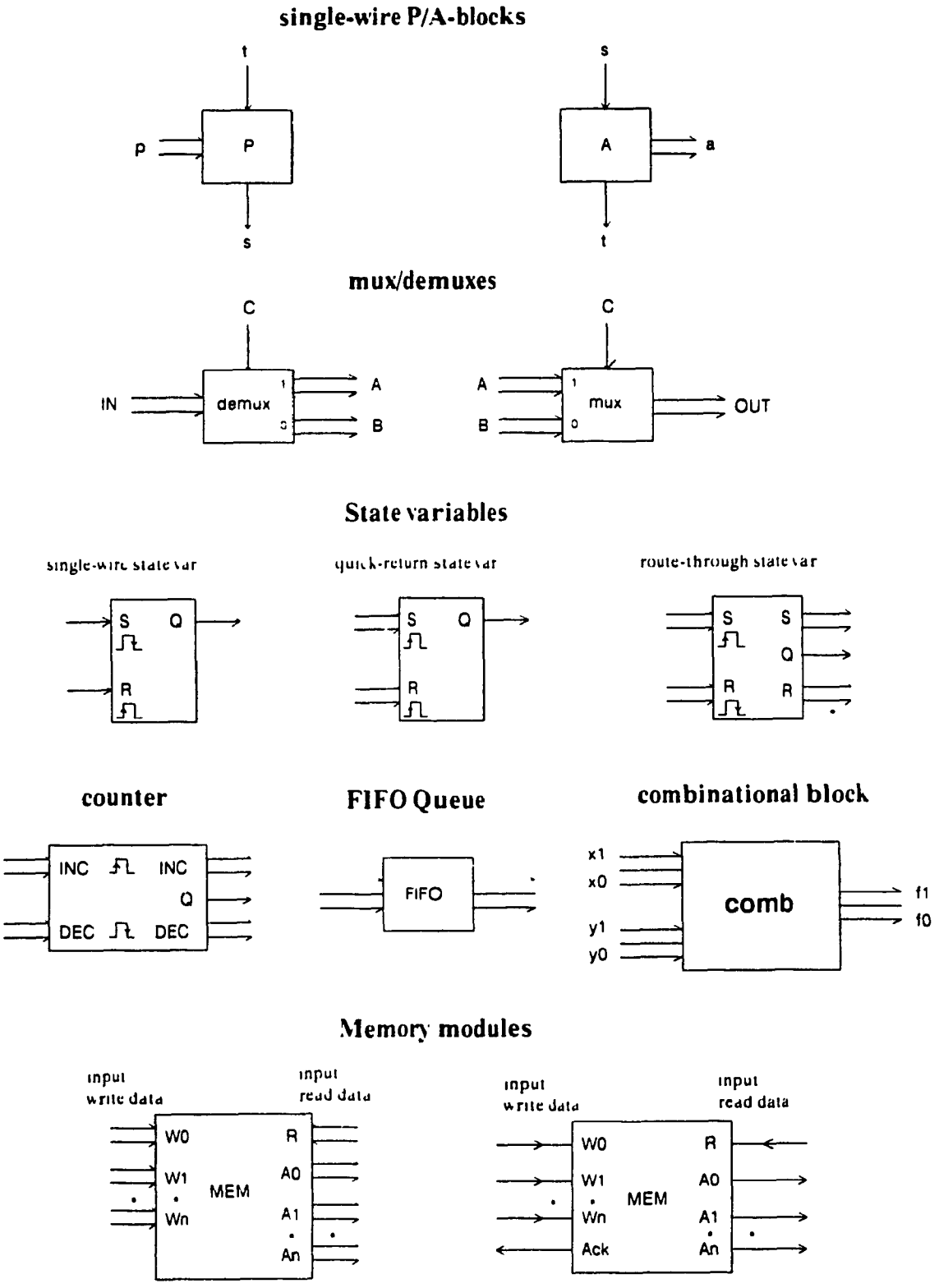



**Figure 6-2: Hybrid building blocks (partially delay-sensitive) and other useful DI building blocks.**

The P/A-blocks, as we have seen, are used for the sequential control of channels (the channel P/A-blocks are shown because they are DI). For internal deterministic choice, there are the test modules which consist of the select block, the test-and-set, test-and-reset, and the alternator. The mutex block is used for internal non-deterministic choice. It is the basic arbitration module, implementing mutual exclusion between two channels. The higher level bit channel operator is the C-block. The C-block synchronizes bit channel data bits together.

Hybrid building blocks and other useful DI building blocks are also provided, as shown in Figure 6-2. The hybrid building blocks are called hybrid because their interface contain both delay-insensitive channels and delay-sensitive single wires. They are provided to allow the user better area efficiency and more flexibility at the cost of some delay analysis. The single-wire P/A-blocks are a direct replacement for the channel P/A-blocks. The mux, demux, and the state variables are used for the internal construction of the test modules. The counter may be used in place of the state variables, changing the output state only after a certain number of increments or decrements. The other useful DI building blocks are larger grain DI modules. Although they can be constructed from the basic set of DI building blocks, they are included here because more area efficient implementations for them exist. This includes the common FIFO queue and buffer, the combinational block for evaluating combinational functions, and the memory modules for reading and writing of data. A more detailed description of all the blocks will be given in the next section.

There are two types of channels used for communication by the DI building blocks:

1) route-through channels
2) quick-return non-shuffled channels

Suppose a module M contains a passive left channel L and an active right channel R. If L is implemented with two wires, [li] and lo, and similarly for R, ro and [ri], then the trace for a route-through channel will be [li+], ro+, [ri+], lo+, [li-], ro-, [ri-], lo-.

Request signals coming in from the left essentially get routed through the module. Quick-return channels on the other hand will have the trace [li+], lo+, [li-], lo-. An in-coming request will get immediately acknowledged by the module.

The DI building blocks are restricted to using only these two types of channels so that the packet model can be used in the composition of the building blocks. Packets can flow through route-through channels undisturbed and the quick-return non-shuffled channels satisfies the packet model's requirement for maintaining discreteness of packets. The restriction of using only these two types of channels greatly reduces the complexity in the design of the system, as we do not have to verify the correctness of shuffled handshake signals at the signal transition level to ensure freedom from deadlock. Deadlock only needs to be considered at the higher packet level.

In the basic set of DI building blocks, the only quick-return channels that exist are in the P/A-blocks. All the other building blocks use route-through channels. Additional quick-return channels will appear when a building block has its output channel shorted by a wire, directly connecting its output request to its corresponding acknowledge wire. This type of connection causes the building block to absorb the packet in the same way that a quick-return channel does, rather than routing the packet through the building block. The source of all packets are from the environment at the input channels of the DI system and also at the output of the A-blocks. The destination of all packets where they will be absorbed are at the quick-return interfaces, namely at the input of the P-blocks and at the shorted output channels.

As an example of how packets are used in the DI building blocks, Figure 6-3 shows how packets flow in the route-through constructs. The fork, join, and mjoin route-through constructs are used to manipulate channels and direct the flow of packets. In the simplest case of one-to-one communication, an active process A sends a packet to a passive process P through simply two wires. To implement one-to-many communication or a broadcast, a fork building block is used. An active process sends a packet through the fork module which will split

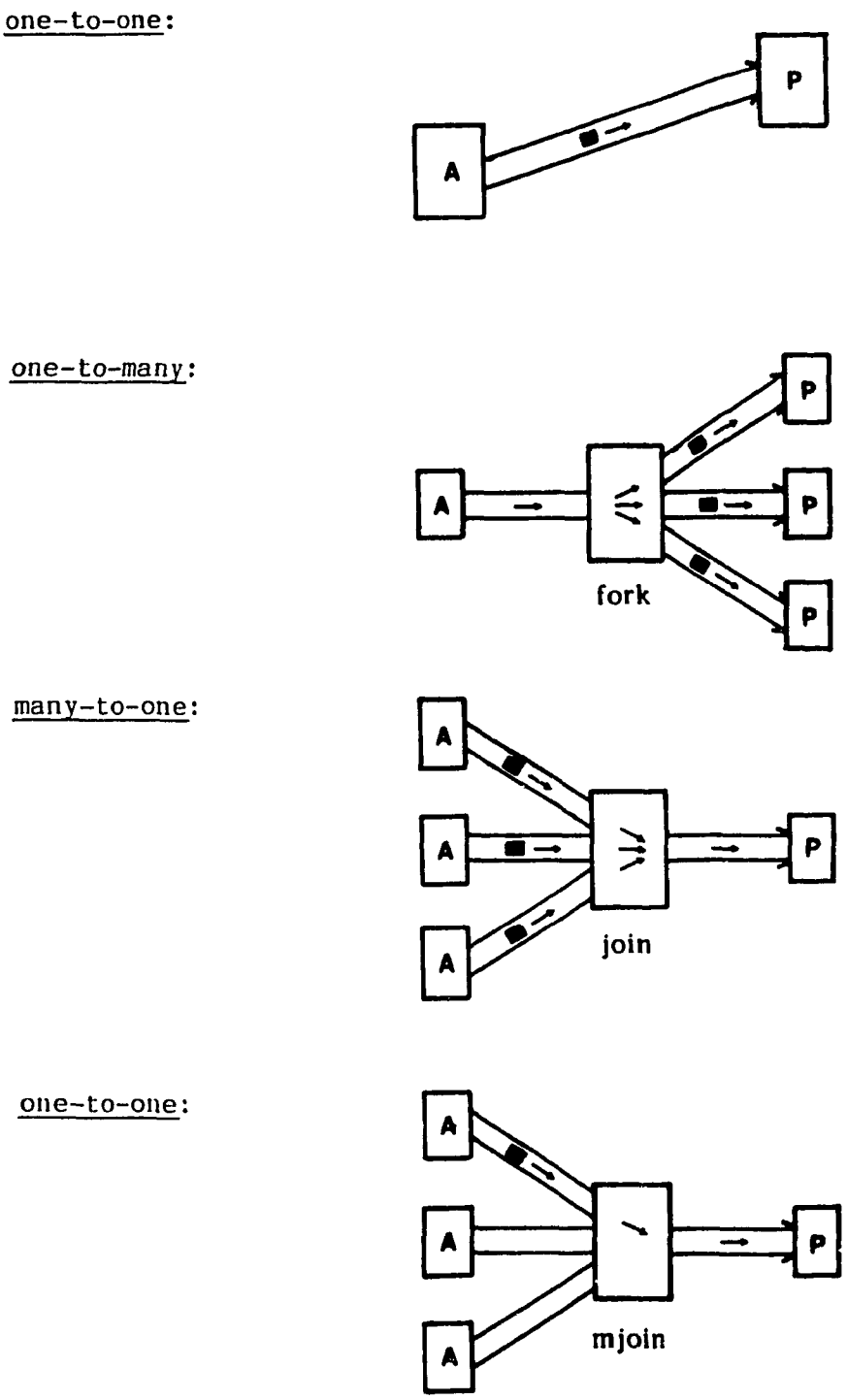



**Figure 6-3: Flow of packets in the route-through constructs.**

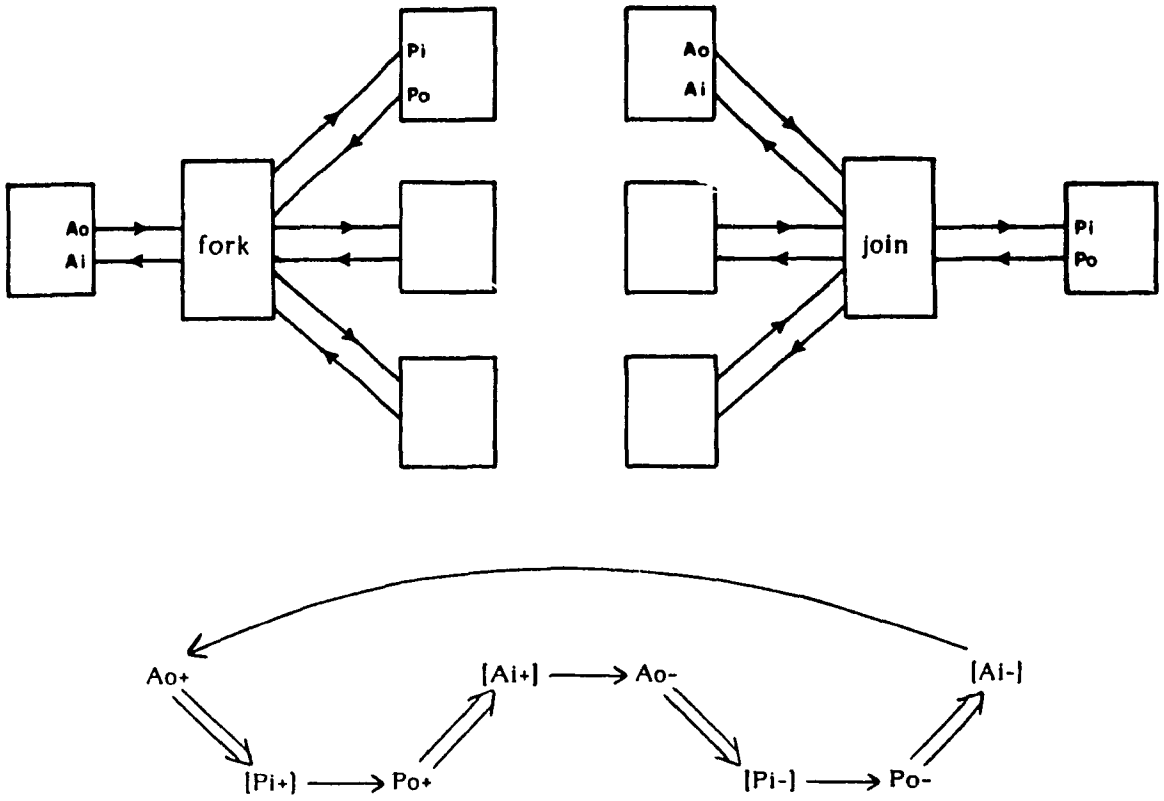



**Figure 6-4: The fork and join at the signal transition level .**

the packet up and send it to a number of passive processes. The join building block does the opposite, joining a number of packets into a single packet. The mjoin building block is a mutually exclusive join or a merge join. It accepts only one packet at a time from a number of active processes and routes it through to a single passive process (the environment of the mjoin as a result of the composition must ensure that mutually exclusive packets are sent to the mjoin). The fork, join, and mjoin all have route-through channels. They are transparent to the packets and routes them through undisturbed.

To justify the fork and join operations on the packets, Figure 6-4 shows the fork and join communications at the signal transition level. The STG with respect

to each active sender A and passive receiver P is identical to that of a one-to-one packet communication through a pair of wires. Since there are many receivers to the fork, the single packet is seen as having forked to all the passive processes, and similarly for the join.

## 6.2 Specification and Circuit Implementations

The detailed description of the DI and hybrid building blocks is organized into two parts, first as formal STG specifications and then as gate level implementations. This distinction is made because there can be many possible implementations which may satisfy the STG specifications of the building blocks. The implementations given are the ones that most obviously satisfy the requirements. The implementations are described down to the logic gate level, so as to remain technology independent; transistor level optimization can be done depending on the technology used, such as CMOS, ECL, GaAs, etc.

The specification of each of the building blocks are described by three different graphs: 1-phase, 2-phase, and 4-phase. The 4-phase graph is the one that is most commonly associated with STG's. It describes the 4-phase handshake protocol used by the two wire channels in the building blocks and also refers to the circuit implementations which use 4-phase handshaking. The 1-phase and 2-phase graphs are used as higher level specifications of the modules. In particular, the 1-phase graph refers to the packet level of abstraction. The 1-phase graph is a channel signal graph (CSG), or alternatively a packet signal graph. Channel signal graphs are required as we move away from wire signal transitions and into the channel signal level. This higher level of abstraction is required as STG's become too complex in describing larger circuits, as will be evidenced, for example, by the C-block. CSG's will become useful for specifying larger delay-insensitive systems which use packets for communication. The 1-phase CSG can be obtained from the 4-phase STG by deleting all the nodes in the 4-phase graph, with the exception of the initial positive edge transition. If the 4-phase graph is divided into its 4 different phases,

the CSG is obtained by deleting the second, third, and fourth phases. Similarly, for the 2-phase graph, the second phase is deleted to obtain the corresponding CSG.

The notation of a CSG is similar to that of an STG, except that signal transitions are replaced by channel events. A DI module will contain passive input channels and active output channels, as opposed to input and output wires. The firing of a transition (vertex node labelled with a channel variable) in a CSG corresponds to the execution of a channel event. The semantics is that once a positive wire transition on the request wire of the channel is seen, the channel variable is said to have fired. The channel variable cannot fire again until the 4-phase sequence is completed. CSG's can be composed together in the same way that STG's can be composed Note that CSG's do not contain the lower level details of wire transitions and so a CSG composition cannot detect deadlock at the wire transition level. In order to have a correct composition, a constraint must be placed on the lower signal transition level, such as the constraints used by the packet model, allowing only the use of non-shuffled and route-through channels. More work is required in clearly defining the syntax and semantics of CSG's and how they can be composed.

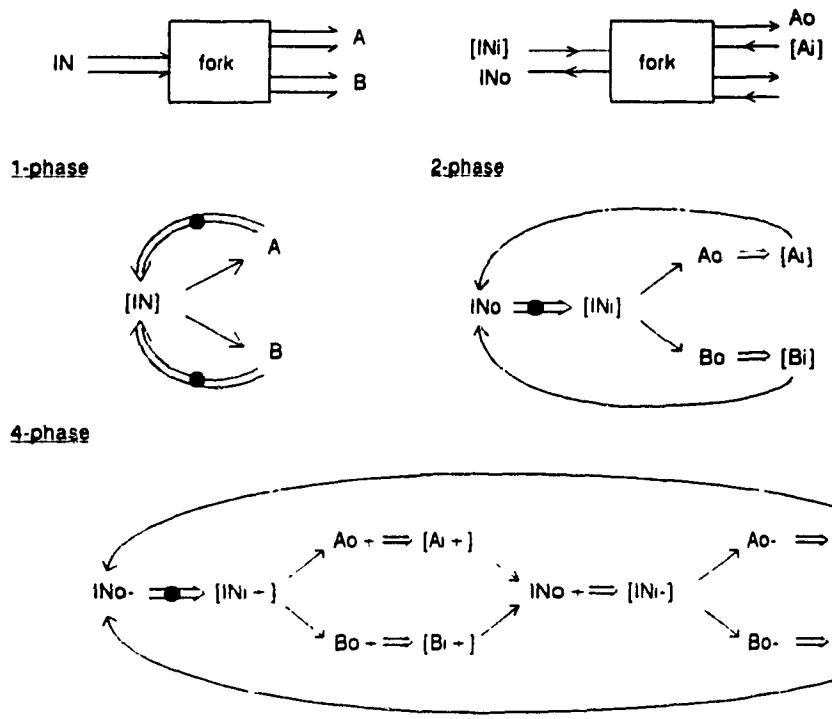
### 6.2.1 Route-through Constructs

The route-through constructs consist of the fork, join, and mjoin. Their specifications are shown in Figure 6-5 and Figure 6-6, and their implementations are shown in Figure 6-15. These building blocks are used for the manipulation of channels. As previously characterized by packets, the fork splits a single packet into many packets, the join combines many packets into a single packet, and the mjoin takes a single packet from many channels and routes it through to a single output.

In the specifications, only the channels of the DI modules are labelled and not the individual wires. To obtain the vertices of the graphs other than for the

1-phase graph, a common notation is used throughout all of the specifications. For example, given a channel CH, if CH is a passive input channel, then its corresponding request and acknowledge wires will be [CHi] and CHo. The "i" indicates that the wire is an input wire to the module and the "o" indicates that it is an output wire. If CH is an active output channel, then its corresponding request and acknowledge wires will be CHo and [CHi] respectively. An example of the explicit labelling of the individual wires is shown for the fork module. The 4-phase graph will contain the usual plus and minus signs to indicate a positive or negative signal transition of the wire. The 2-phase graph does not contain the plus and minus signs; its wire transitions are implied to alternate between positive and negative transitions The initial marking for all of the specifications have the tokens placed on the environment double arrows. This indicates that the modules are in their stable states and are awaiting inputs from the environment.

The implementations for the fork, join, and mjoin are fairly simple. The join is implemented with a many input C-element; this follows how C-elements are commonly used to join or synchronize wire signals (a many input C-element can be implemented with a tree of 2-input C-elements or as a dedicated gate at the transistor level). The fork is also implemented with a many input C-element, but its use is slightly different. The fork block is intended to fork out channels. Forks are usually associated with a single wire forking out into many wires. But in the case of a channel fork, a C-element is required to join all the acknowledge signals from the forked out channels to ensure that all of the output channels has received the broadcast A simple OR gate cannot ensure this. Although the C-element implementations are simple, it is how the C-element is used that is important. One of the goals of this thesis is to reduce the design complexity of constructing large systems. An important method is to use simple structured blocks which have well defined behaviors. Using C-elements at the wire level without attaching the proper semantics of a fork and join leads to ad hoc and unclear designs. The structuring of the DI building blocks is a method of organization and does not necessarily mean reduced area efficiency.



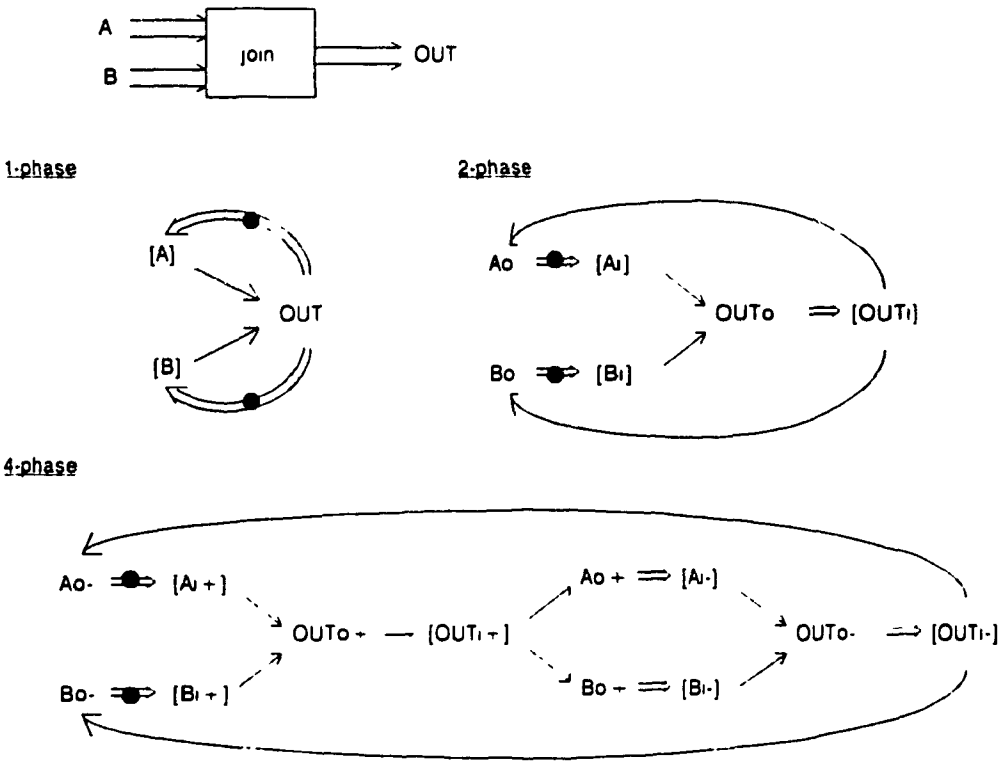
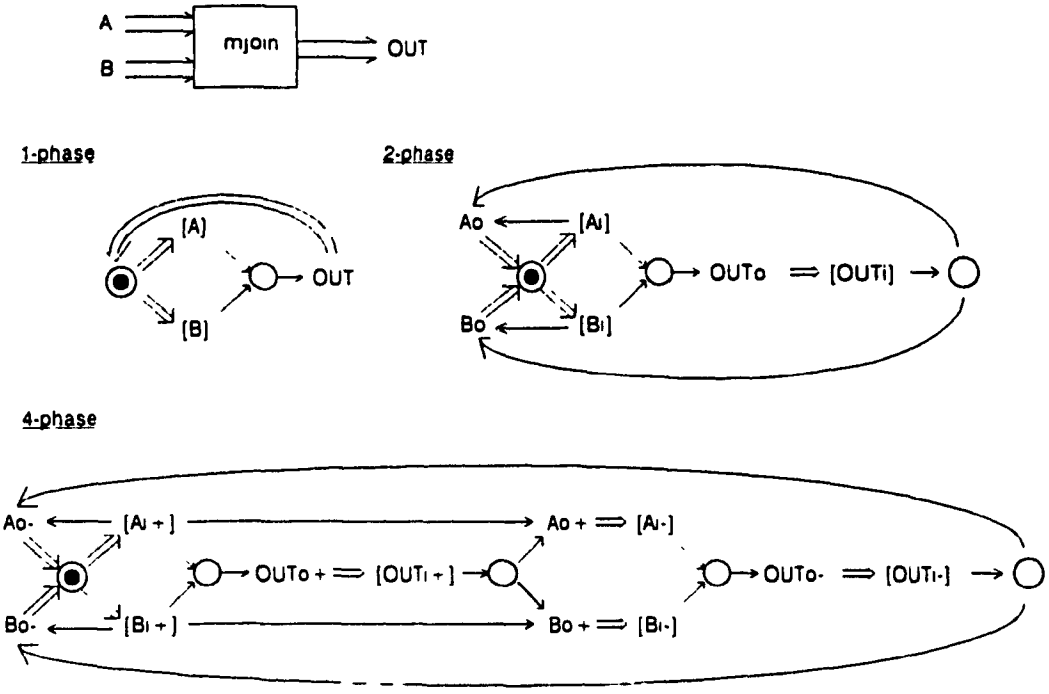




**Figure 6-5: Specifications for the fork and join.**



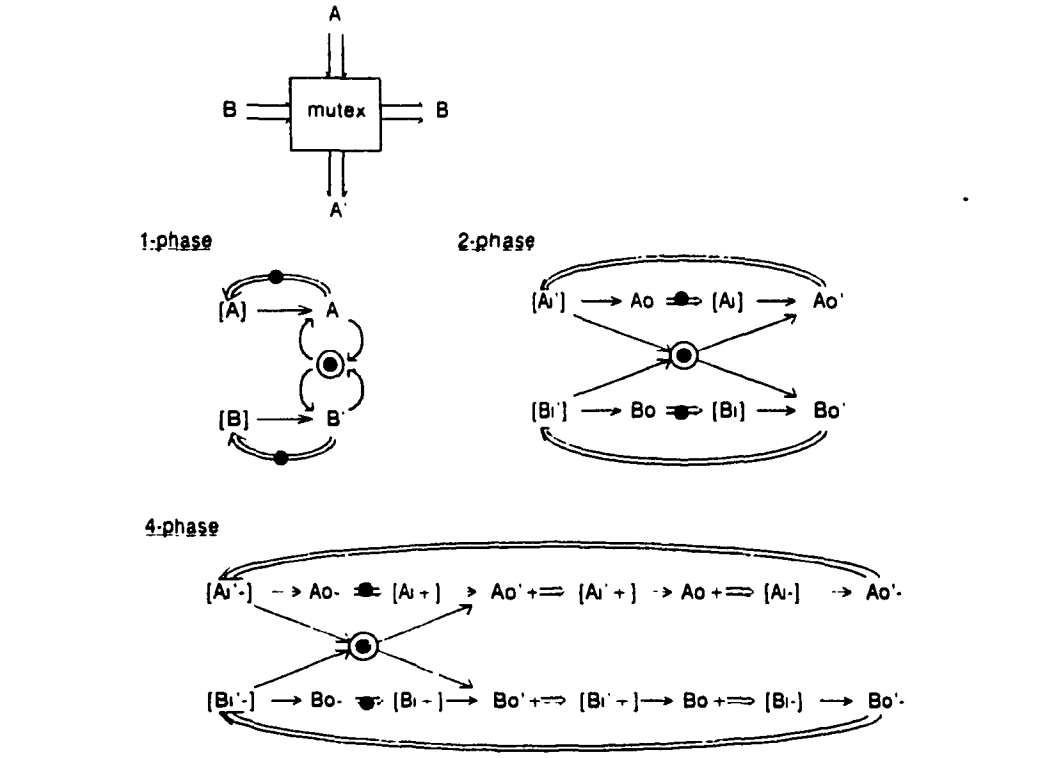
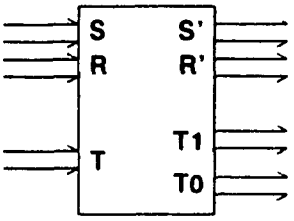



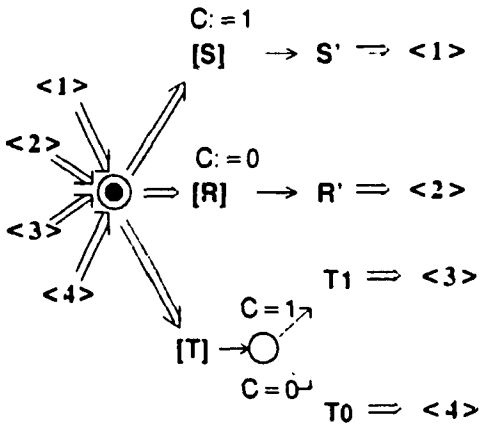

**Figure 6-6: Specifications for the mjoin and mutex.**

**select**

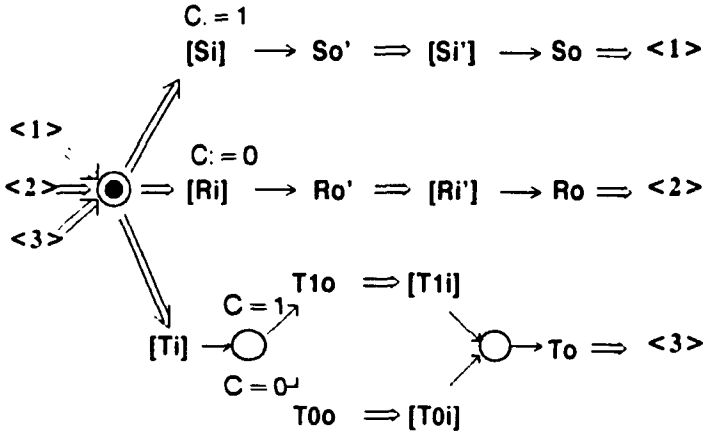



(The connectors < n > are used to connect arrows for simplifying the drawing of the graph.)

**1-phase**




**2-phase**




**4-phase**

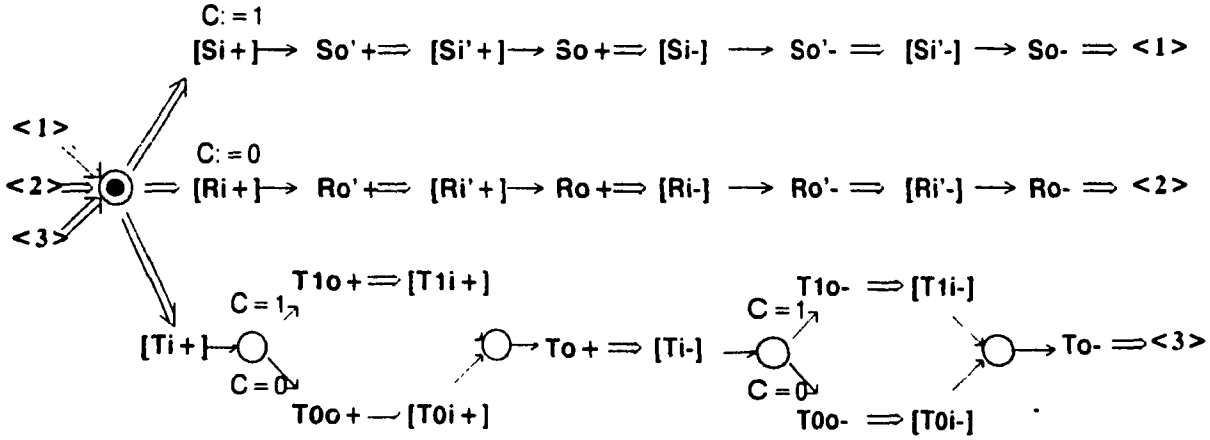
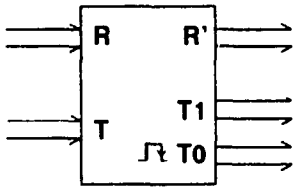
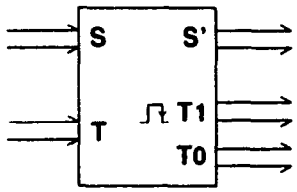



**Figure 6-7: Specifications for the select.**

**test-and-set**

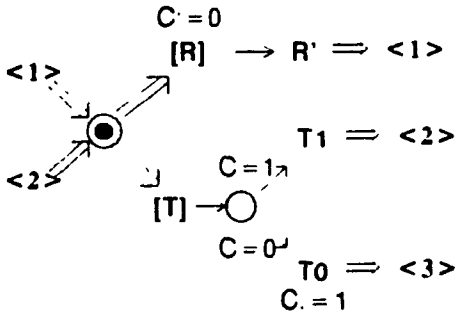



**test-and-reset**

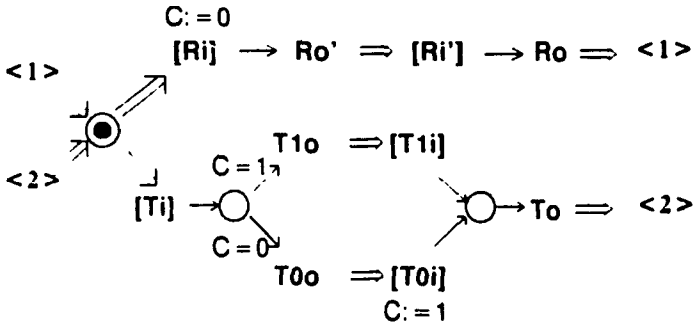



**(similar specification to test-and-set)**

**1-phase**




**2-phase**




**4-phase**

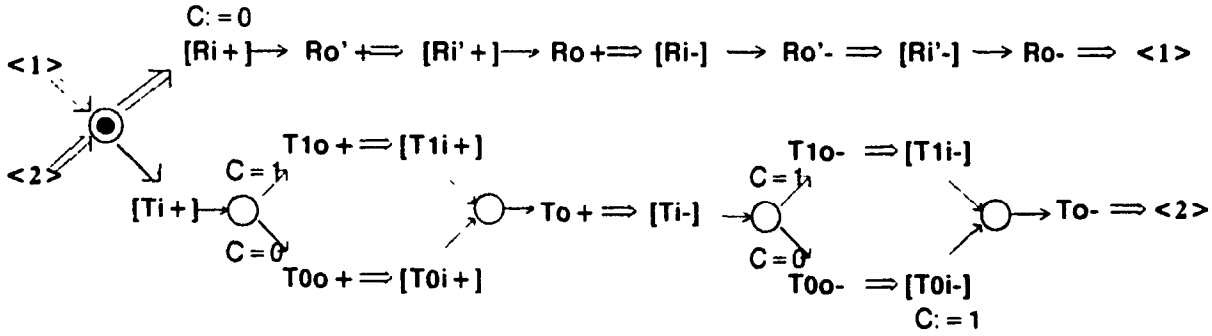



**Figure 6-8: Specifications for the test-and-set.**

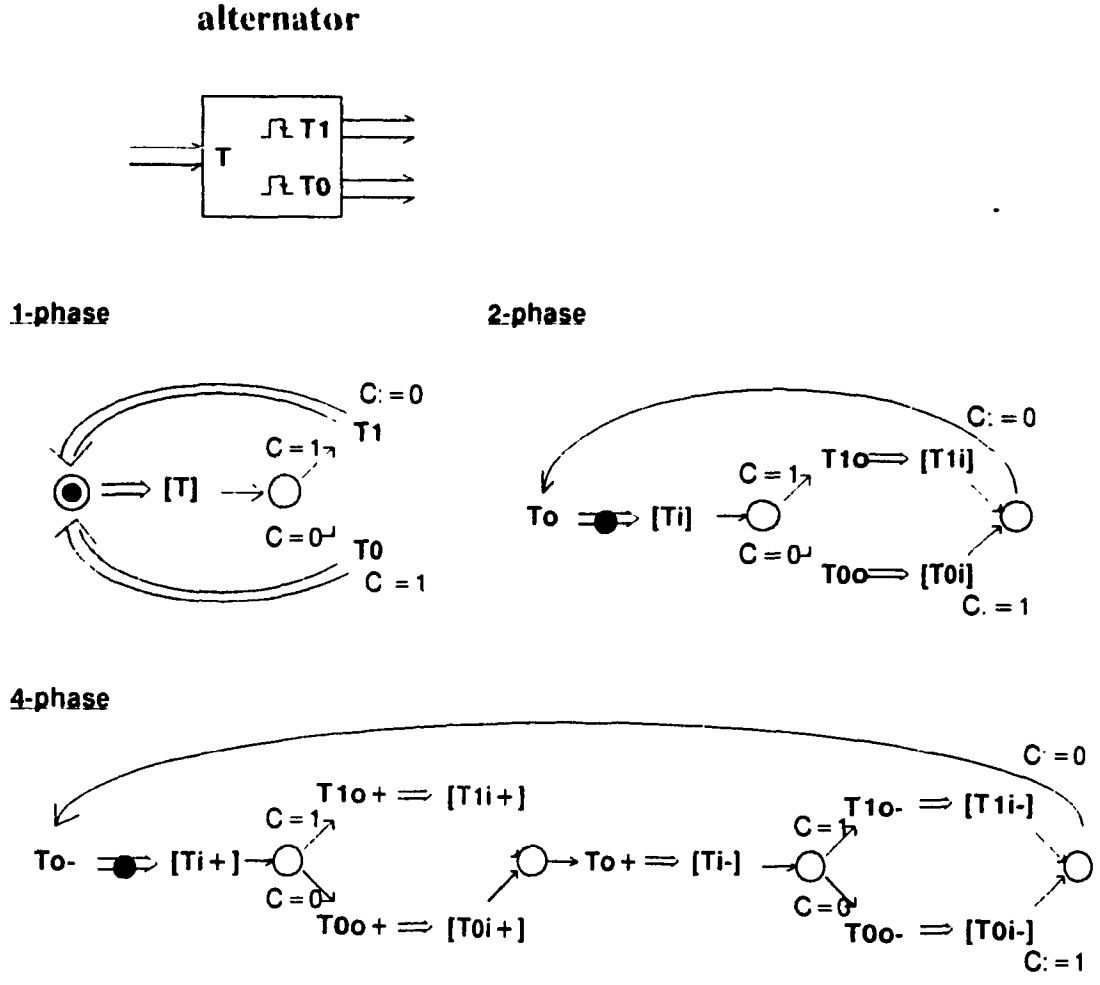
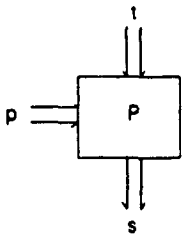
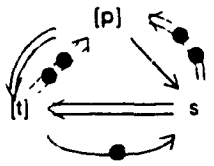


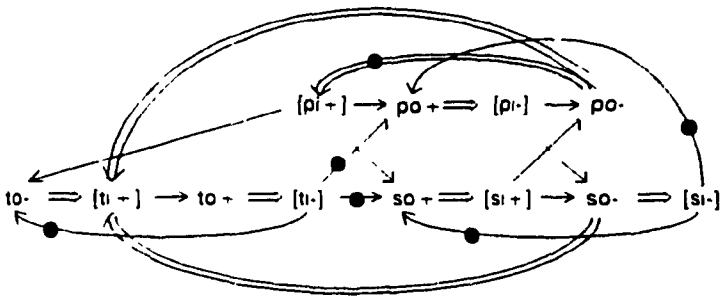

**Figure 6-9: Specifications for the alternator.**

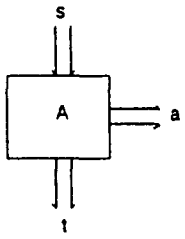



1-phase

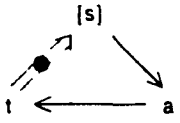



4-phase




1-phase




4-phase

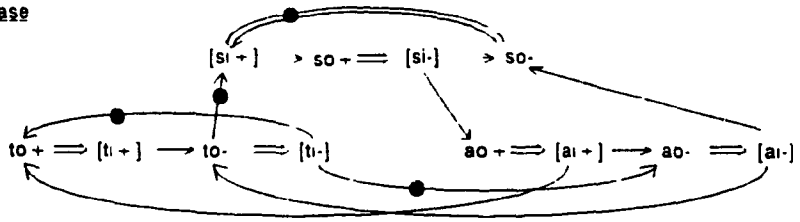
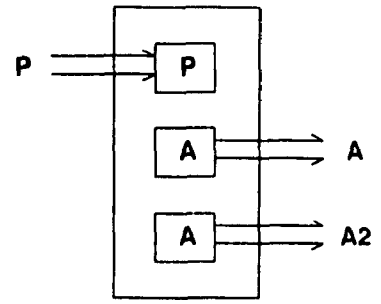
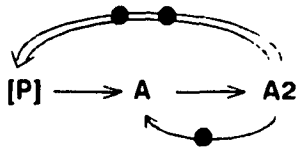


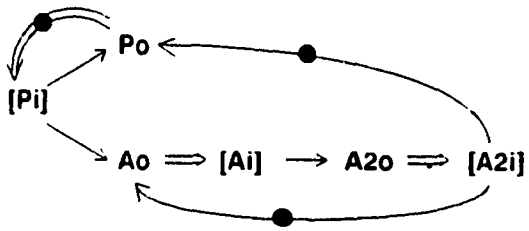

**Figure 6-10: Specifications for the channel P-block and the channel A-block.**




1-phase




2-phase




4-phase

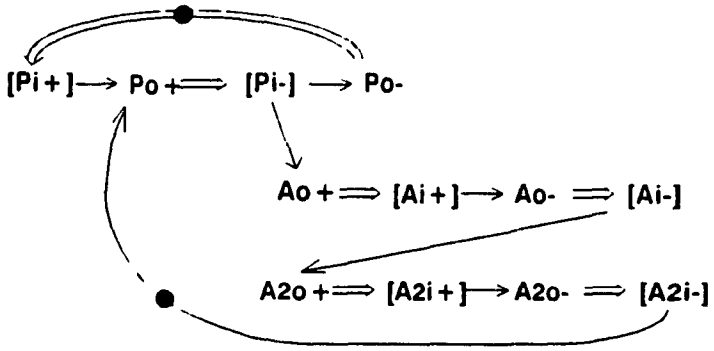
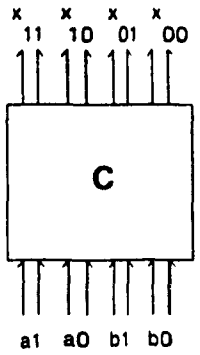


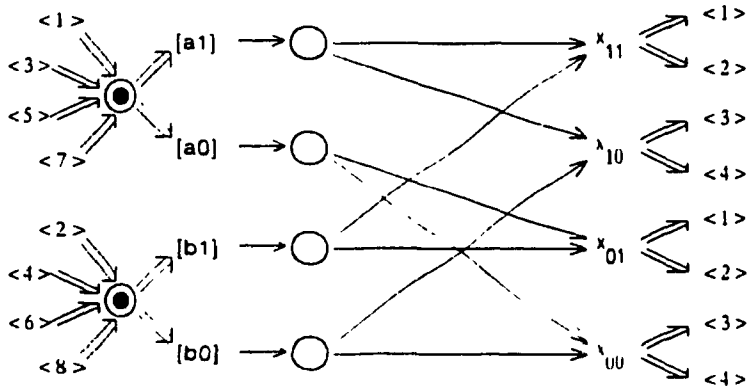

**Figure 6-11: Specifications for a sequential module as a result of the composition of one P-block and two A-blocks.**




**C-block**

(The connectors <n> are used to connect arrows for simplifying the drawing of the graph )

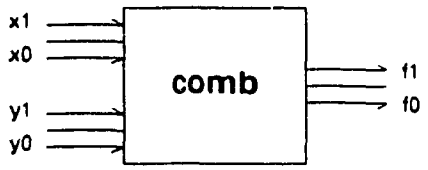
1-phase



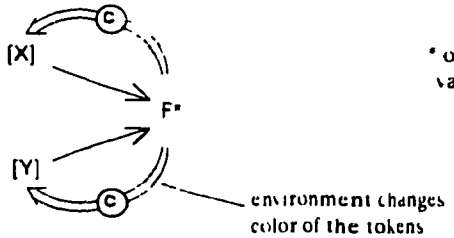

2-phase
+
4-phase

: simple but messy (all route-through channels)

**combinational block**




1-phase




* output channel F sends correct value depending on the function

**Figure 6-12: Specifications for the C-block and the combinational block.**

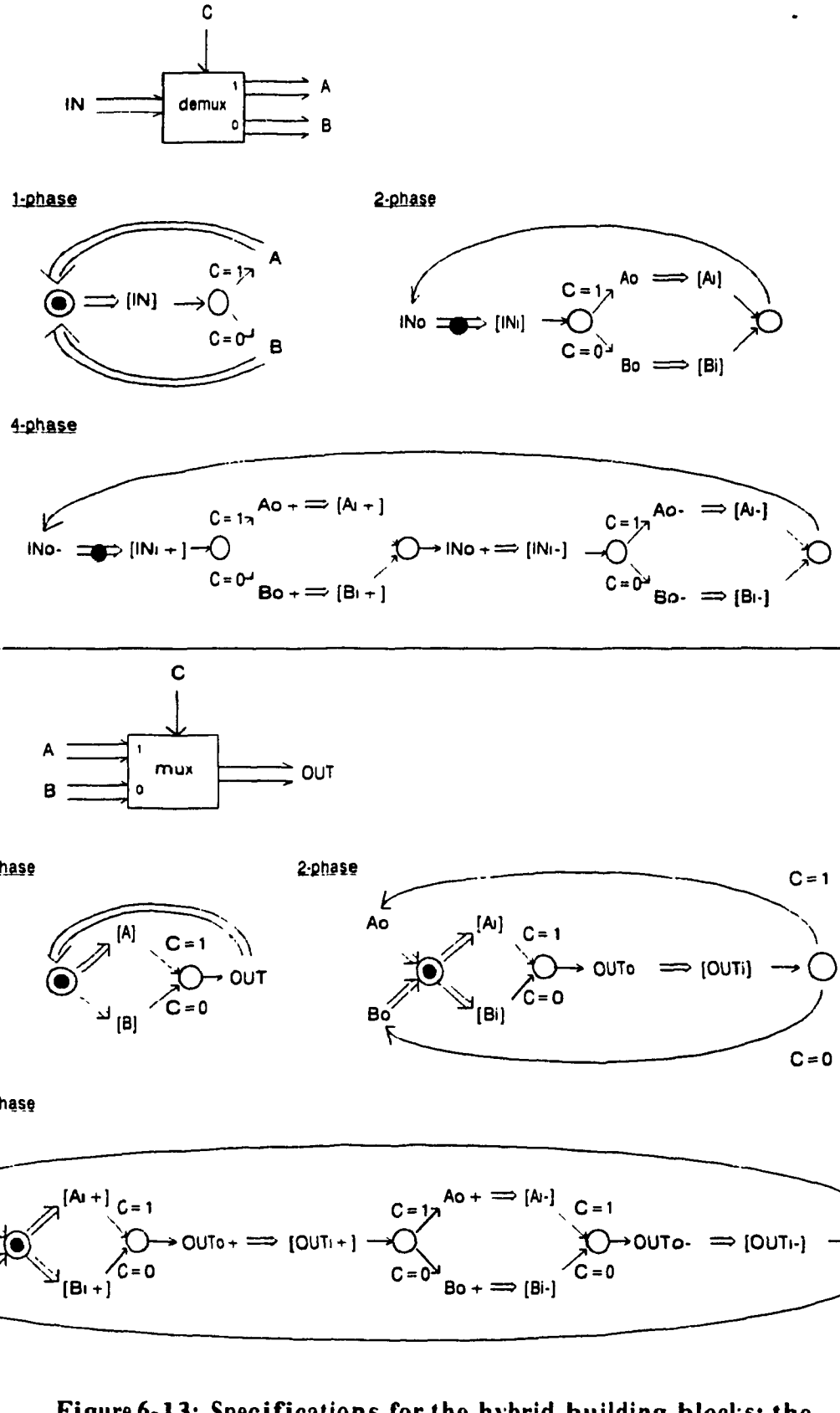
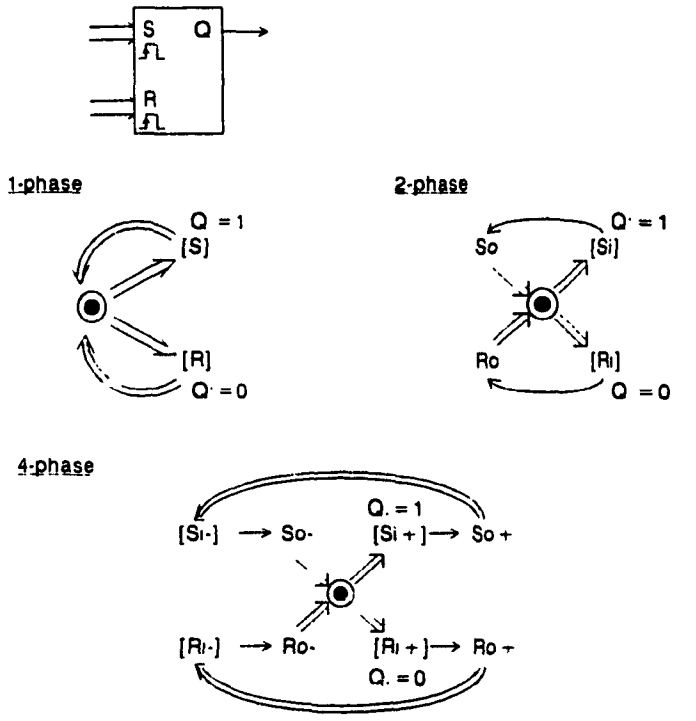



**Figure 6-13: Specifications for the hybrid building blocks: the demux and mux.**



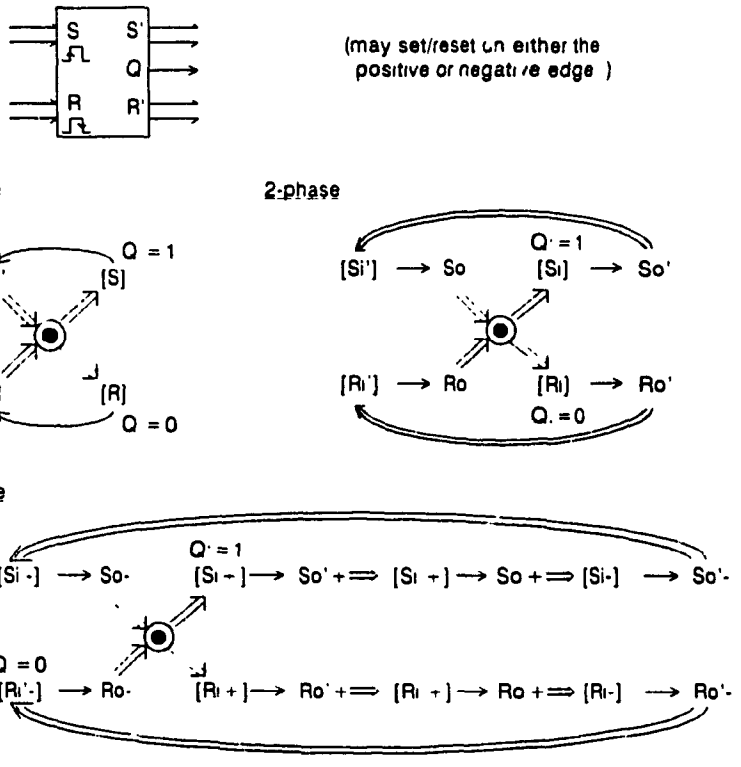




**Figure 6-14: Specifications for the hybrid building blocks: the quick-return and route-through state variables.**

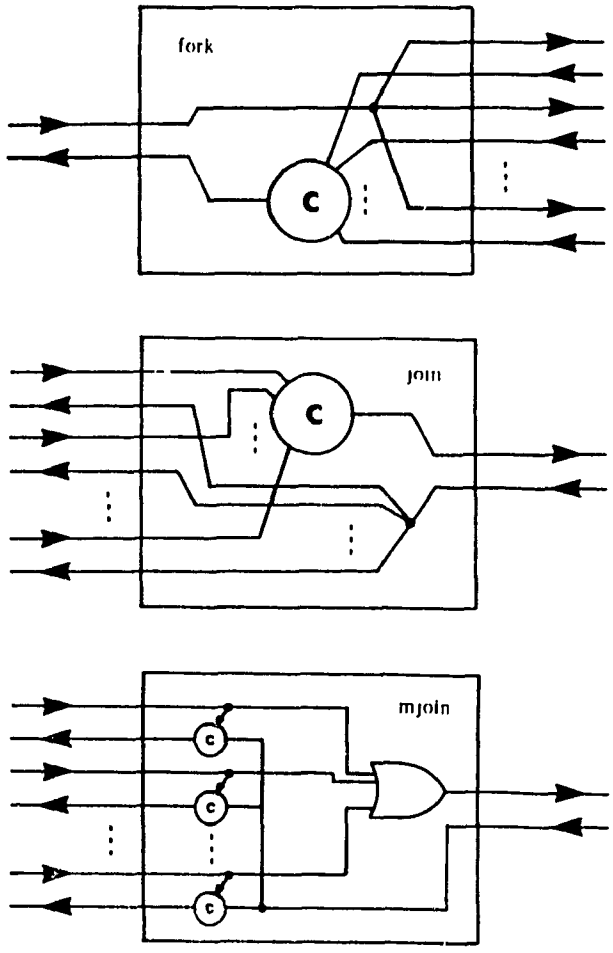



**Figure 6-15: Implementations of the route-through constructs.**

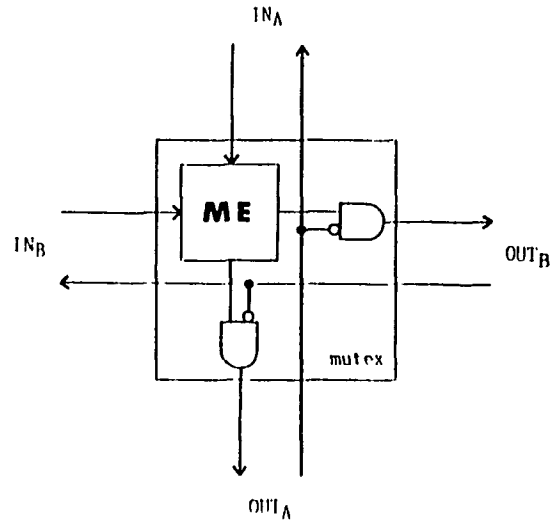
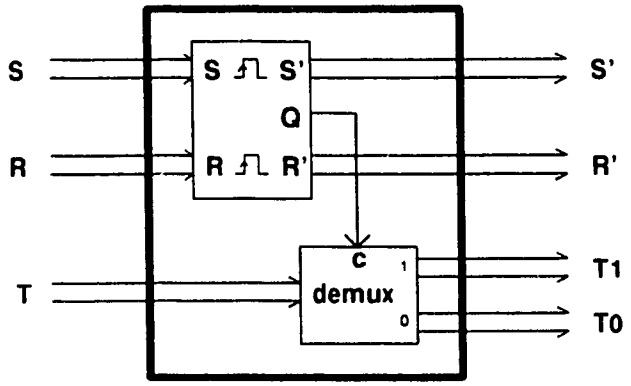
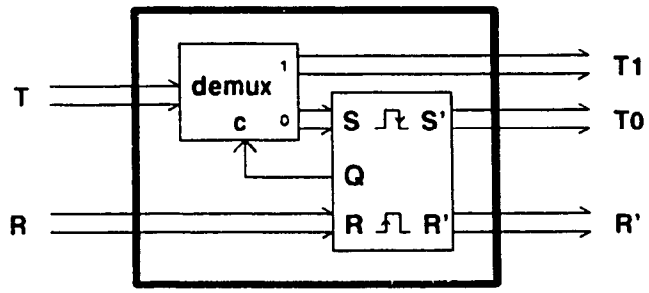



**Figure 6-16: Implementation of the mutex block.**

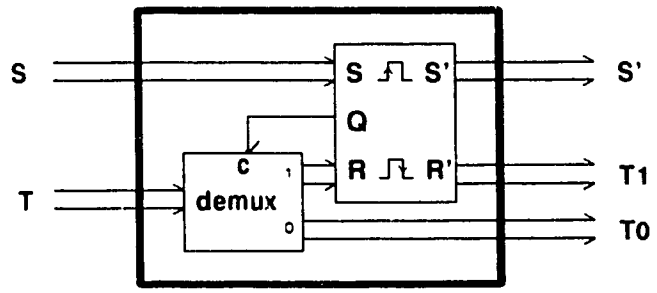
select




test-and-set




test-and-reset




alternator

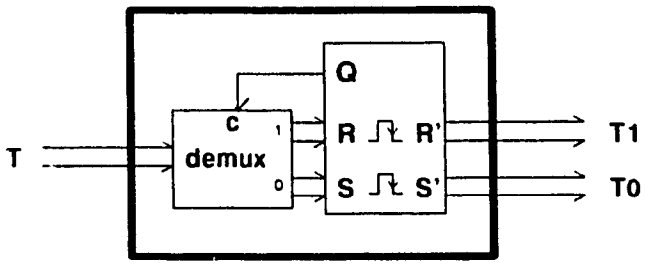



**Figure 6-17: Implementation of the test modules.**

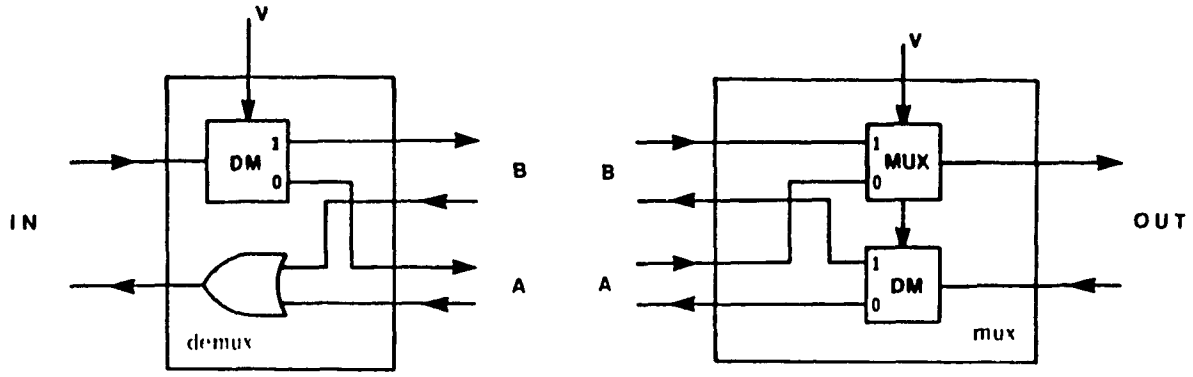



**Figure 6-18: Implementation of the demux and mux.**

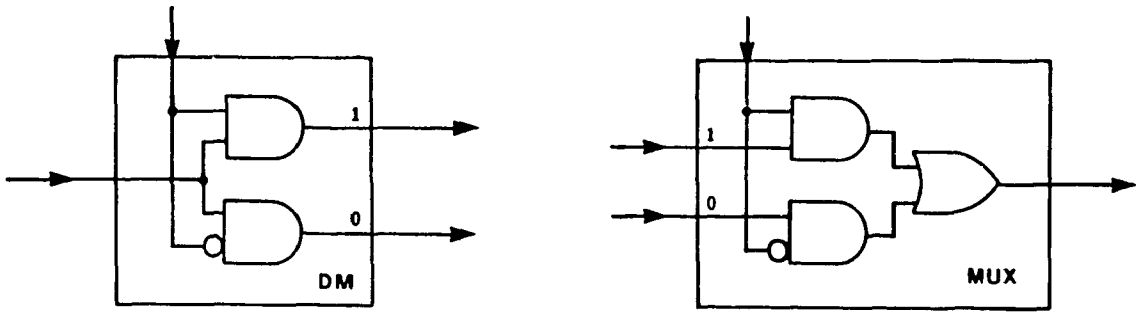



**Figure 6-19: Implementation of the single wire demultiplexer and multiplexer.**

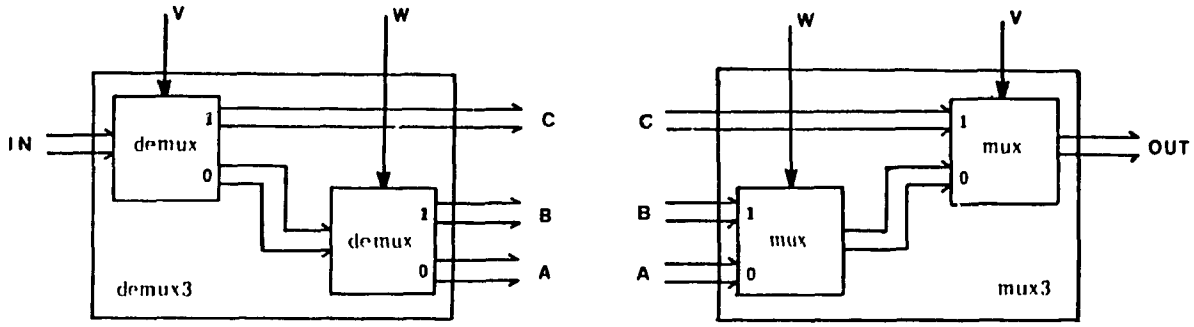



**Figure 6-20: Extension of the demux and mux.**

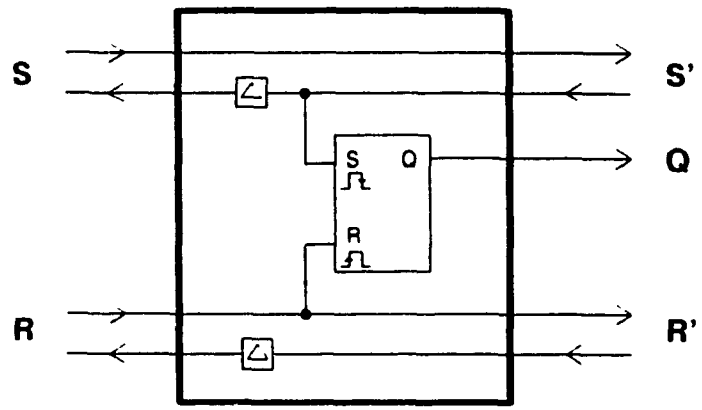



**Figure 6-21: Implementation of the route-through state variable.**

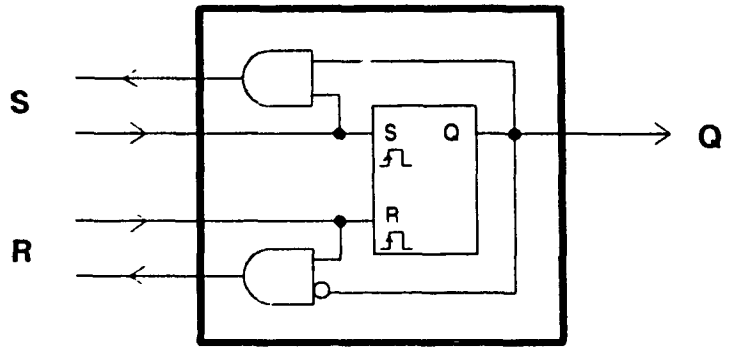



**Figure 6-22: Implementation of the quick-return state variable.**

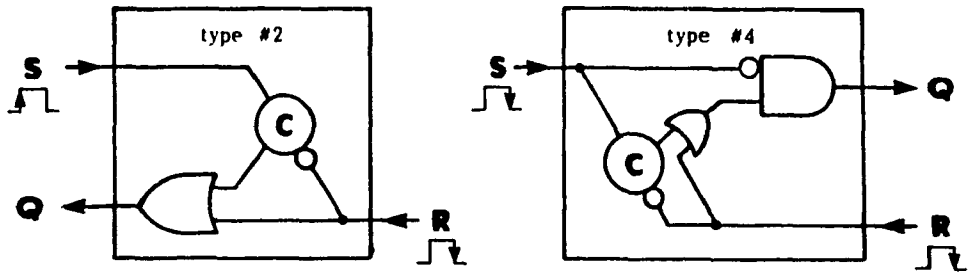



**Figure 6-23: Implementation of type #2 and type #4 single-wire state variables.**

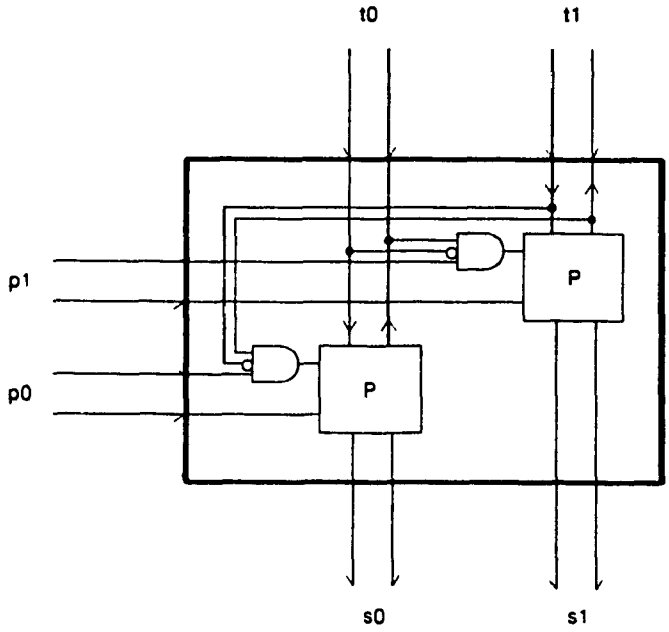



**Figure 6-24: Extension of the P-block to ensure mutual exclusion on a bit channel.**

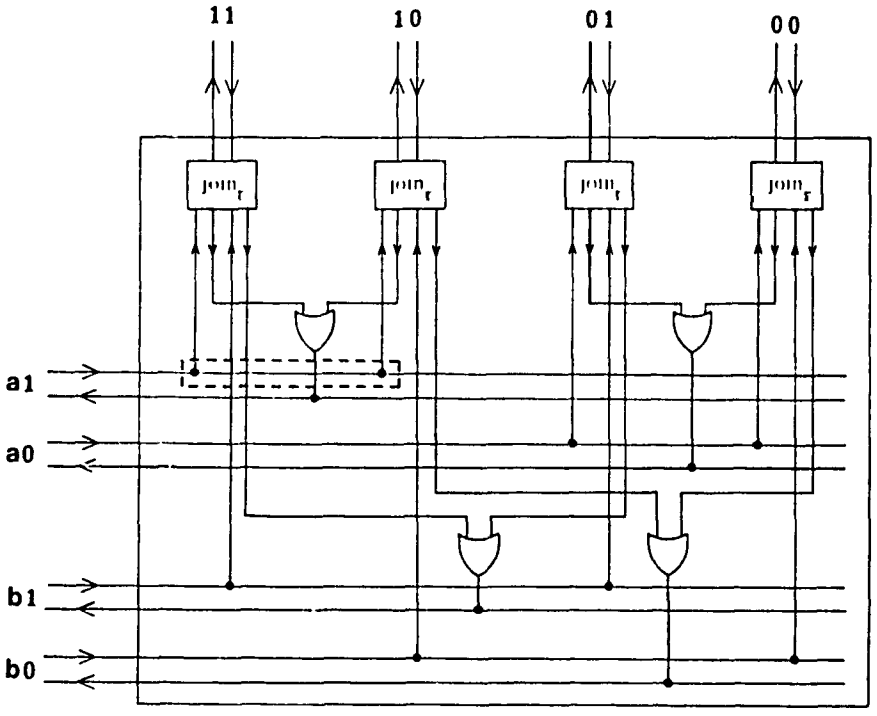



**Figure 6-25: Implementation of the C-block.**

# C-elements = # rows in function
= 2^N , where N = # input variables

size of C-elements = N

| x | y | f |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

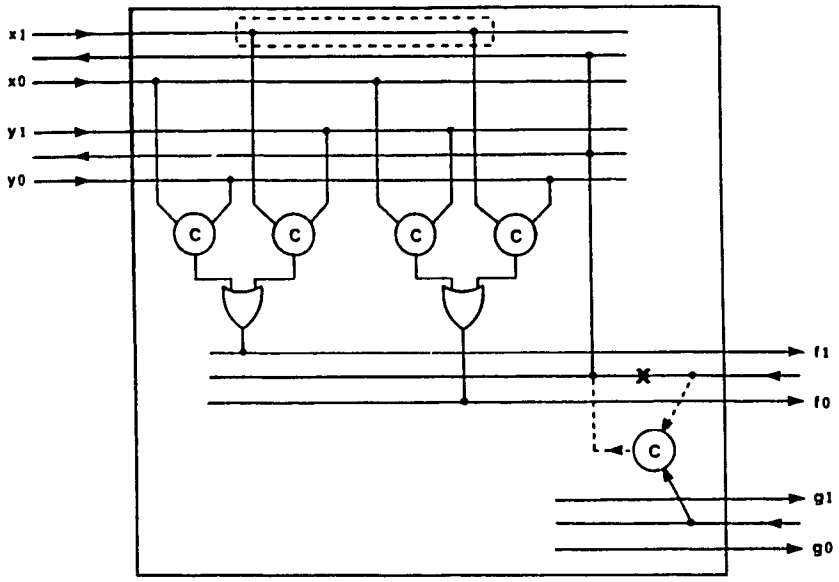



**Figure 6-26: Implementation of the combinational block.**

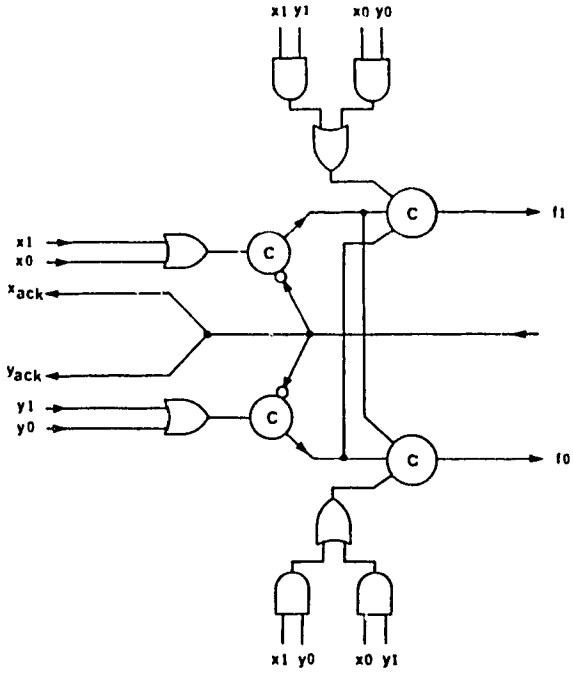



**Figure 6-27: Alternative implementation of the combinational block.**

The mjoin is used to join a number of mutually exclusive channels. It is analogous to a wire merge, which is commonly implemented with an exclusive-OR gate. Note that mutual exclusion of the input channels must be ensured by the environment. This is indicated in the specifications by a shared place sitting on environment controlled double arrows. The implementation of the mjoin uses one large OR gate to join the mutually exclusive requests entering the block A C-element is required for each input channel because the returning acknowledge signal from the single output channel must know which of the input channels to acknowledge. Only the single input channel that made the request should be acknowledged and not the ones that did not make a request. Note that the fork, join, and mjoin circuit implementations also satisfy the 2-phase specifications, even though the intended use of the DI building blocks is for 4-phase operation.

### 6.2.2 Mutex Block

The mutex block is used for ensuring mutual exclusion between two channels, which is the special case of arbitration between two (channel) users. Its specification is shown in Figure 6-6. Two channel signals (requests) can come into the mutex block at the same time on channel A and channel B. The winner of the arbitration gets its request routed through the module while the loser must wait for its request to be acknowledged. The channels are route-through channels; for example, if user A wins the arbitration, the path A to A' through the module appears as simply two straight wires.

The specifications of the mutex block contain an internal non-deterministic choice. This is easily seen by the shared place sitting on the single arrows, indicating that the choice is controlled by the module. Referring to the 4-phase diagram, two requests coming in on channels A and B will cause [Ai+] and [Bi+] to fire concurrently. Because of the single token on the shared place, either Ao'+ or Bo'+ may fire. If channel A wins the arbitration, then Ao'+ fires and deletes the token from the shared place thereby disabling channel B. The channel A

token then proceeds through its request-acknowledge sequence, which is also the same as the sequence for a route-through channel. The token is returned to the shared place when [Ai'-] fires, corresponding to the final negative acknowledge signal returning on channel A'. The waiting channel B can then immediately fire Bo' + .

Note that this arbitration is considered to be fair because B will never be starved. The token on channel B sitting between [Bi + ] and Bo' + is waiting and can immediately fire once A returns the token to the shared place. A, on the other hand, cannot immediately make a request to compete for the token in the shared place because its returning token must incur a delay from [Ai'-] to Ao', through the environment and module arrow delays. As long as this delay is larger than the delay for B to recognize that the token has returned to the shared place (which is usually the case in actual circuits), then the arbitration is fair. If A and B continually make requests, then grants will alternate between A and B; this is called strict fairness

The implementation of the mutex block is shown in Figure 6-16. It is based on a single wire mutual exclusion element (ME). The ME arbitrates between high requests on two incoming wires. The winner of the arbitration causes its corresponding acknowledge wire to go high. The ME is released when a low transition comes in on the original request wire. An example circuit implementation of such an ME can be found in [Seit80]. To implement the channel mutex block two extra AND gates are required. They are used to ensure that the release of the ME does not occur prematurely. The mutex block should only be released on the final low acknowledge of the completed channel signal. The final low acknowledge enables the AND gate and lets a request from the waiting opposite channel go through. Note that the opposite channel has already won the arbitration through the ME and is only waiting to be enabled at the AND gate; this ensures fair arbitration between the two channels. The implementation of this mutex block satisfies the 4-phase specification and not the 2-phase one.

### 6.2.3 Test Modules

The test modules consist of the select block, the test-and-set, test-and-reset, and the alternator. These modules are used in internal deterministic choice. They are used in the selection of channels. In the simplest case, these modules have two states; each state selects between one of two channels. The specification of the select block is shown in Figure 6-7. The environment is constrained to sending in either one of three input channel signals into the block, S, R, and T. This is indicated in the graphs by the shared place sitting in the double arrows. Note that the connectors are used to connect arrows; they are only used as a notation to simplify the drawing and complexity of the graph and do not change the semantics. The select block contains two internal states, which is specified by the control variable C. If a channel signal comes into T, then a selection is made between two output channels (specified by the internal shared place). If the present state is C = 1, then the output channel T1 is selected, and if the present state is C = 0, then the output channel T0 is selected. These channels are route-through channels To change the state C, a channel signal is sent into S to assign C: = 1 or into R to assign C: = 0. In the 4-phase graph, the state is changed on the positive edge of the input request signal. As previously stated in the chapter on STG's, the introduction of the state variable C is used to reduce the complexity of the graph; the graph can be explicitly drawn out to form this control operation

The test-and-set block in Figure 6-8 is similar to the select block but does not include the set input S. The setting of the control variable C is done at the same time as the input (test) of the channel signal at T (hence the name test-and-set). If the present state is C = 0, a channel signal coming into T will get routed through the output channel T0. Upon finishing the channel signal (at the returning low acknowledge signal [T0i-] in the 4-phase graph), the control variable is set to C: = 1. Any subsequent channel signals into T will get routed through T1. The state is reset only upon a channel signal through R. The setting

of the control variable C in the test-and-set can only be done on the final negative edge (returning low acknowledge [T0i-]) of the channel signal. This is indicated by the negative edge pulse next to T0 in the test-and-set block. The state cannot be changed upon the positive edge request because the channel signal is still active. The selection of the alternate output channel T1 cannot be done until the 4-phase sequence on T0 has completed, which means when the final low acknowledge (negative edge) returns to the block.

Note that the test-and-set block cannot be constructed from the select block by connecting its output channel T0 to the input channel S. This self-loop to simulate the simultaneous routing through of a channel and setting of the control state (test-and-set) cannot be done because the self-loop creates a concurrent input into the select block on the channel inputs S and T. This would violate the specification of the select block of mutually exclusive inputs to S, R, and T. A specification of a select block that would allow for such a self-loop must take into account the different states of the 4-phase sequence on the channels of the self-loop and would be very difficult to implement. Hence a separate test-and-set block is used.

The test-and-reset block is similar to the test-and-set, except that the control state is tested and reset rather than set The implementation of the test-and-reset can be done by simply exchanging the T1 and T0 outputs of a test-and-set and reversing the interpretation of the state Therefore, the test-and-reset is used only for notational convenience.

The alternator in Figure 6-9 is similar to the test-and-set in that it alternates between test-and-set and test-and-reset of the block. A channel signal coming into the T input alternates between being routed through T1 and T0. No R or S inputs are required (although they can be explicitly included). The alternator is included as a separate basic DI building block because it cannot be implemented out of the other DI building blocks.

The implementations of all the test modules are shown in Figure 6-17. They are implemented out of the partially delay-sensitive hybrid building blocks. Two building blocks are used in each of the implementations, a demux and a route-through state variable. The channel demux block is required to make the selection of the output channels T1 and T0 upon an input from T. The route-through state variable is required to record the state and control the demux. The state variable sets and resets on either the positive or negative edge. The negative edge is used whenever a simultaneous test and set/reset is required, so that the state of the state variable can be changed upon the finishing low returning acknowledge signal to the block. The ordinary S and R inputs set on the positive edge, as soon as the high request of a channel signal comes in. Although the change of state in the block may occur at any time during the S and R channel handshake sequence (such as the case of a negative edge), changing the state as soon as the channel signal arrives is the most efficient.

### 6.2.4 Hybrid Building Blocks

The hybrid building blocks presently include the mux/demux, the state variables, the counter and the single-wire P/A-blocks. The mux/demux, state variables and counter are used in the implementation of the test modules. The single-wire P/A-blocks are used as more area efficient replacements for the channel P/A-blocks. These building blocks are called hybrid because they are partially delay-sensitive; their interfaces contain both delay-insensitive channels and delay-sensitive single wires. The existence of this delay-sensitivity implies that there are certain constraints when these blocks are composed together. The delay analysis/requirements for the single-wire P/A-blocks was shown in Section 4.5. The delay requirements of the other blocks are essentially that the pulse widths to set/reset the state variables are sufficient and that their Q outputs reach the destination blocks they are controlling in time before the next input to the controlled block arrives. The use of hybrid building blocks allows for more

flexibility in constructing higher level DI building blocks and the possibility of area-time optimization.

**Mux/demuxes:**

The mux/demux blocks select the channels in the test modules. The specifications are shown in Figure 6-13 and their implementations in Figure 6-18 to 6-20. These blocks operate in the same manner as the traditional multiplexers and demultiplexers, but are extended to operate at the channel level. Depending on the control input C, different channels are selected. These channels are of the route-through type. The demux block is used in the test modules, but the mux block can be used in its place to perform the operation of multiplexing of channels. As with the traditional single-wire multiplexers and demultiplexers, these blocks can be extended to control a larger number of channels, either through a tree of blocks or as a single dedicated block. This extension would imply the use of more than one state variable for control.

**State variables:**

The state variables are used for memory, recording the state of a single bit value. Their outputs directly control the mux/demux blocks. There are three types of state variables: (1) the route-through, (2) the quick-return, and (3) the single-wire state variable. The quick-return and route-through state variables have channels as interfaces. Their specifications are shown in Figure 6-14 and their implementations in Figure 6-21 and 6-22. The route-through state variable may either set or reset on the positive or negative edge of a signal transition on a channel wire, while the quick-return one sets and resets on the positive edge. The route-through state variable is the more general one in that it can implement a quick-return state variable by shorting its S' and R' outputs with a wire. The advantage of the quick-return one is that it does not require delay-elements

(denoted by a triangle in a box), and so does not need delay analysis to satisfy the pulse width requirements of the internal state variable.

The internal implementation of the route-through state variable uses a single-wire state variable. These are asynchronous R-S flip flops, but are extended to set/reset on the positive or negative edge of a pulse. There are four possible types of single-wire state variables:

type #1 : S on positive edge / R on positive edge
type #2 : S on positive edge / R on negative edge
type #3 : S on negative edge / R on positive edge
type #4 : S on negative edge / R on negative edge

The type #1 state variable is implemented as the traditional cross-couple connected NOR gates which are positive edge sensitive. The type #3 state variable has the same implementation as the single-wire A-block, whose circuit implementation has been shown previously (Q = request of the A-block, R = acknowledge, S = sequencing wire). Figure 6-23 shows implementations for the type #2 and type #4 state variables. Here, we assume the C-elements have the greatest gate delay in order for correct operation. Note that some inputs do not allow two consecutive set/resets, in which case, their intended operation is to have the R-S inputs alternate. These are only possible implementations; there should be others that allow unrestricted set/resets.

The state variables only record a true or false boolean value. It would be useful to have building blocks which can record more than boolean variables, such as the counter block that was shown in Figure 6-2. The interface of the counter block is configured in the same way as the more general route-through state variable. Its Q output is also meant to control a mux/demux block. It has two route-through channels, an increment (INC) channel and a decrement (DEC) channel. These two channels operate similar to the R-S channels of the route-through state variable, except that the output Q does not change with every set or reset. The Q output becomes high only after a certain number of increment channel signals enter the counter block. Otherwise, the Q output will be low. For example, a counter block of size five initialized to a zero count will set only

after the fifth INC channel signal. The decrement channel works oppositely, decrementing the count. As in traditional synchronous counters, there can be different possibilities. The counter can be preset to any desired value at system initialization. The counter can be configured to rotate back to the zero count after an increment past the maximum value, thereby resetting the Q output; alternatively, it can be configured to ignore subsequent increment channel signals, maintaining the Q output high after going past the maximum count. A more general counter block would also include additional R and S channels, corresponding to asynchronous set and clear operations.

The counter block is considered as a larger grain block. It contains many states as compared to the state variables. Its use implies that the resulting DI module will be larger. Although the counter operation can be implemented out of finer-grain state variable based DI modules, use of the counter block will provide better area efficiency. The implementation of the counter block would be similar to that of the ones in synchronous systems. Essentially, the channels are used to provide a pulse which is then used as a clock pulse to increment/decrement the counter (the pulse mode of operation will be further discussed in chapter 8).

### 6.2.5 P/A-blocks

The P/A-blocks have already been discussed previously. They are used to create a sequence of active and passive channel events. The specifications of the DI channel P/A-blocks are shown in Figure 6-10. Their implementations were shown in Figure 3-2. The p channel and the a channel are non-shuffled with respect to one another. The s and t sequencing channels, on the other hand, *are* shuffled with respect to the p and a channels. The sequencing channels are used only to connect the P/A-blocks together and are not used to connect with the other DI modules.

Note that the initial marking of all the P/A-blocks are not the same when they are connected to create a sequential module. As stated in chapter 3, the state variable in the initial P-block of the sequence must be initialized to a one to enable the sequence. This means that the first P-block will have the wire 'to' high, as indicated by the location of the token in the P-block specification. The initial marking for the other P-blocks will have 'to' as low.

An example specification of a sequential module consisting of one P-block and two A-blocks is shown in Figure 6-11 (note that the P/A-block implementations satisfy only the 4-phase and 1-phase specifications). From the 1-phase specification, it can be seen that the P-block input has a quick-return (with a buffer size of 1). In the initial marking, two tokens sit on the environment double arrows. When the first channel signal comes in, [P] fires and removes the first token from the environment. At this point, the A transition does not have to fire yet. Since there is another token sitting on the environment arrow, [P] can immediately fire again. This second immediate firing indicates that the first channel signal had a quick-return input. The environment does not have to wait before sending a second input request into the module. At this point the environment contains no more tokens. It has to wait for the sequential process to fire channel signals A and A2 before it can send in another request. In the 4-phase graph, it can be seen that one complete 4-phase handshake sequence on the P channel can be completed without having to wait for A channel acknowledges (corresponding to the quick-return). The second input request (from the firing of [Pi+]) will have to wait for the handshake sequence on the A and A2 channels to complete before Po+ is enabled and the P-channel handshake sequence allowed to continue.

### 6.2.6 Extension to the Bit Channel Level

Up to now, the DI building blocks control only single channel events. A single two wire channel indicates only the existence of a control event and does not convey any data information. To describe data values, the basic two wire channel

must be extended. To do this, more than one channel can be used. For example, to describe a binary data bit, two channels can be used: one channel to indicate a binary "1" value and a second channel to indicate a binary "0" value. This pair of channels taken together results in a binary bit channel.

It is fairly easy to extend the DI building blocks to the bit channel level. The fork, join, and mjoin can be made to manipulate bit channels by simply putting two blocks together, side by side. A bit channel can then fork or join through these pairs of blocks. This can also be done for the mutex block. By using two mutex blocks, one can be used to arbitrate the "1" channel and the second to arbitrate the "0" channel. To extend the test modules to the bit channel level, only the T selection channels need to be extended. The R-S set/reset channels can remain as single control channels. To extend the T channels, the internal implementation using a mux/demux block can be doubled. One mux/demux block can be used to route through the "1" bit and the other to route through the "0" bit. The single state variable then controls two blocks instead of one, the Q output of the state variable connecting to the two C control inputs of the mux/demux blocks.

In the case of the P/A-blocks, doubling them up will result in two sequencing channels (or sequencing wires). Because the P-blocks use quick-return, some extra circuitry is required. To ensure mutual exclusion of channel signals on the bit channel, the first pair of P-blocks at the head of the bit channel sequence requires the extension shown in Figure 6-24. The two AND gates are required to retain mutual exclusion of the "1" channel bit and the "0" channel bit. They are used to disable an incoming request from the opposite channel until the present bit has been completely processed. Otherwise, both the "1" and "0" channels will become active, which is meaningless. Originally, mutual exclusion of the two channels is ensured by the environment. But because there is a quick-return in the P-block, an error may occur if two channel signals come in too quickly on opposite channels. For example if a channel signal comes into p1, the "1" bit channel, it will immediately be acknowledged because of the quick-return. But

before the "1" bit sequence is finished processing its sequence of blocks, a channel signal may come into p0 and activate the "0" bit sequence. To ensure mutual exclusion, the AND gates disable the incoming request on the opposite channel whenever the present channel is active. It is enabled once the sequencing channel t returns to its original state after the sequencing loop is finished.

This type of connection to ensure mutual exclusion also occurs in quick-return FIFO queues. Because of the increased area usage, it may be desirable to implement the bit channel P-block as a separate optimized block. Alternatively, the P/A-blocks may be converted into non-quick-return blocks and a separate FIFO queue element be used to implement the pipelined operation. (Note that all the other DI building blocks use route-through channels and do not contain pipelining. Buffering of the building blocks by inserting FIFO buffers is meant to be a separate operation. To increase area efficiency, separate buffered fork, join, etc. building blocks can be used.)

### 6.2.7 C-block

Once single channels are extended to the bit channel level, a separate block will be required to handle bit channels. The C-block is used to synchronize bit channels. Its specification is shown in Figure 6-12 and its implementation in Figure 6-25. Two bit channels come into the block, a and b. Depending on the value of the data bits, a single output channel signal is given (the C-block also uses route-through channels). The C-block is similar to a bit channel join.

The implementation uses four joins to join the four possible output conditions. Each join is responsible for joining two input channels, either a one or a zero. The OR gates are required to send back the acknowledge signals to the appropriate channels that made the request. This is a speed-independent implementation. Isochronic forks must be assumed, such as that indicated by the dashed region. If the internal wires have significant delays, then delay

compensation must be done to balance the delay of the two forked wires or by putting a delay element on the acknowledge line to make sure that both wires of the fork have reached the same potential before sending back an acknowledge signal.

Note that the C-block cannot be implemented out of the lower level DI building blocks. We cannot simply use four forks forking out to four joins to simulate the four output conditions. This is because channel signals on a bit channel arrive mutually exclusively. A normal fork cannot be used to fork the input channels because it will wait for all its output channels to make an acknowledgement. Since the bit channels occur mutually exclusively, the forks will wait forever to receive an acknowledge. A connection such as the OR gates shown in Figure 6-25 is required.

### 6.2.8 Other Useful DI Building Blocks

Although the basic set of DI building blocks should be sufficient to create all desired DI modules based on single channels, the resulting circuit may not be area efficient. To create more area efficient circuits, it is possible to create dedicated optimized modules for a specific application. Larger grain modules can be used rather than the finer grain DI building blocks. The tradeoff is more design complexity and decreased layout freedom of the circuit. An automated compilation method would require that only a fixed set of modules be used and would usually not allow for this type of hand optimization.

Some useful DI building blocks were shown in Figure 6-2. They include the FIFO queue element, combinational block, and memory modules. These can be implemented out of the basic DI building blocks, but at the cost of increased area usage. The FIFO queue is a common element used for buffering and pipelining. It can be implemented out of the P/A-blocks. Since there exist many possible more efficient implementations of FIFO queue elements, it would be desirable to use a separate dedicated block. As well, it is possible to use 3-wire bit channels

rather than 4-wire bit channels as shown on the FIFO queue. A single acknowledge wire can be used to acknowledge both the "1" and "0" bits, rather than having two separate acknowledge wires for both bits. Using three wire channels can decrease the wiring area, but may increase the gate area used to manipulate it. It is possible for the DI building blocks to be based on 3-wire channels or other higher level channels, rather than the basic 2-wire channels (generalized channels and conversion of channels will be discussed in the next chapter).

Another useful larger grain DI building block is the combinational block. Its specification is shown in Figure 6-12 and implementation in Figure 6-26. The specification uses colored data tokens operating at the bit channel level. The colored tokens are used to describe the change in data value of the packet on the bit channel as a result of evaluating the combinational function. Such a higher level graph is more useful than a signal transition graph, as higher level circuits using data channels are used (more work is required in defining its usage). The lower level implementation of the combinational block uses 3-wire route-through channels. C-elements are required to synchronize the two input data values and the OR gates are used to send the proper output request to the output channel. Any combinational function can be evaluated. As well, the number of input variables and output variables can be extended. The combinational block is not included in the basic set of DI building blocks because it can also be implemented using a C-block, with its outputs mjoin'ed together to obtain the desired function. An alternative implementation of the combinational block is shown in Figure 6-27.

The other useful DI building block that was shown in Figure 6-2 was the memory module. Two versions of it are shown, one using single channels and another using an n + 1 wire channel. Data is written into the module through the W channels/wires, and data is read by sending in a request and receiving the data value on the A acknowledges. The memory module can be implemented out of a network of select blocks. The n + 1 wire version can be obtained by

doing channel conversion shown in the next chapter. It is also possible to extend the memory modules by using different data channel encodings. At this point, it can be seen that there can be many different higher level DI modules. It is up to the designer to decide whether to create an optimized dedicated module, rather than building it from finer grain DI modules. Since the DI building blocks are meant to be used as the assembly blocks in an automated silicon compiler, it would be preferable to limit the modules to a fixed basic set.

# Chapter 7

## Generalization and Conversion of Channels

In this chapter, a generalization of the basic two wire channel is presented. Here, we concentrate on only totally delay-insensitive channels. Extending channels to include more than one request or acknowledge wire produces different data encodings, as well as different types of channels, such as read, write, and bidirectional channels. Conversion circuits are shown which converts read and write channels to and from the basic two wire channels of which the DI building blocks are based upon.

### 7.1 Generalized Channels

A single two wire channel by itself represents either the existence or non-existence of a control signal. In order to represent data values or more complex control information, the basic channel consisting of only a single request and acknowledge wire must be extended to include either more request wires, more acknowledge wires, or both.

The simplest method of encoding data is to use a binary bit channel, as seen previously, in which one channel represents a binary "1" bit and a second channel represents a binary "0" bit. Another commonly used bit channel consists of 3 wires rather than 4, two request wires to represent the binary value and a single combined acknowledge wire for both bits. The former type of channel is referred to as 2n wire signalling, where n is the number of bits to represent. A single 2-wire channel is used to represent each bit. The other type of channel is referred to as n + 1 wire signalling, one request wire for each bit plus a single acknowledge wire. Each of these bits occur mutually exclusively. To represent parallel data which are commonly used on data buses, a number of these bit channels can run in parallel to produce *byte* channels.

Any generalized channel C will contain a set of request wires $R = \{r_0, r_1, r_2, ...r_n\}$ and a set of acknowledge wires $A = \{a_0, a_1, a_2, ... a_n\}$. The channel C is described by the following:

$$w = b(n + m)$$

where w is the total number of wires in C,
b is the number of bit channels in parallel,
n is the number of request wires in a single bit channel,
m is the number of acknowledge wires in a single bit channel.

In the case of 2n wire signalling, we have $b = 1$ for a single bit channel, and $m = n$, the number of acknowledge wires equal to the number of request wires. This leads to $w = b(n+m) = 1(n+n) = 2n$. For $n + 1$ wire signalling, $b = 1$ again and $m = 1$ for one acknowledge wire, which gives $w = b(n+m) = 1(n+1) = n+1$.

If $n = 1$, this gives a single two wire channel, $n = 2$ gives a binary bit channel, $n = 3$ gives a ternary bit channel, $n = 4$ gives a quaternary bit channel, etc. If $b > 1$, then bit channels are in parallel. The total amount of numbers that the channel C can represent is $n^b$ while using $b(n + m)$ wires. Fixing $m = 1$ for $n + 1$ wire signalling gives the least number of wires for inter-module communication, but using 2n wire signalling may lead to less hardware if the internal module implementation is based on single channels.

Varying the number of acknowledge wires m and request wires n can lead to different types of channels. In particular, the same acknowledge wire does not have to be dedicated for acknowledging a fixed set of request wires. Different acknowledge wires may acknowledge the same request depending upon the state of a process.

There are three possible types of channels:

1) write data channel
   (active sender - passive receiver)
2) read data channel
   (active receiver - passive sender)

3) bidirectional data channel

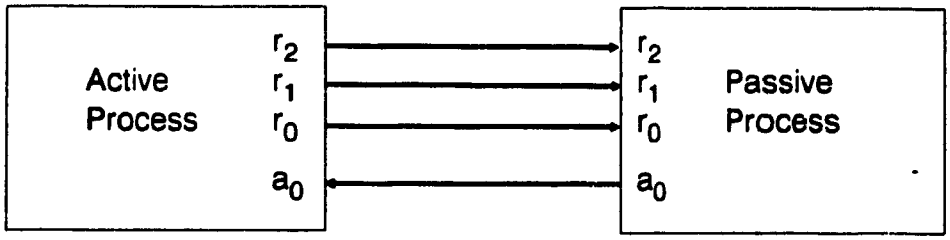
(active sender/receiver - passive receiver/sender)

The write data channel (or data send channel) is the most commonly used. It is used for sending data from an active process to a passive process, the active process being the initiator of the request. The 2n and n + 1 wire signalling are examples of write data channels Figure 7-1(a) illustrates this for $b = 1$, $n = 3$, $m = 1$, $R = \{r_0, r_1, r_2\}$ and $A = \{a_0\}$. Data is sent on the request wires and acknowledge wires are dedicated to acknowledging a fixed set of request wires.

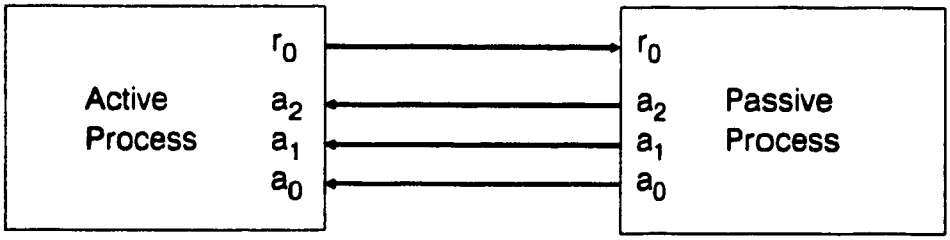
The read data channel (or data receive channel) is used for reading data values. Figure 7-1(b) shows the case for $b = 1$, $n = 1$, $m = 3$, $R = \{r_0\}$, $A = \{a_0, a_1, a_2\}$. In this type of channel, an active process sends a request (request has no encoded data) to the passive process and the passive process returns data in the form of different acknowledge signals. This type of channel is used to read data from passive processes such as stacks and memory modules (Write data channels are used to write into them).

The bidirectional data channel is a combination of both a read and a write data channel. It can both send and receive data. Figure 7-1(c) shows the case for $b = 1$, $n = 3$, $m = 3$, $R = \{r_0, r_1, r_2\}$, $A = \{a_0, a_1, a_2\}$. The active process sends data on the request wires to the passive process and the passive process returns data on the acknowledge wires. The returned data on the acknowledge wires will depend on both the input request data and the state in which the passive process is in.

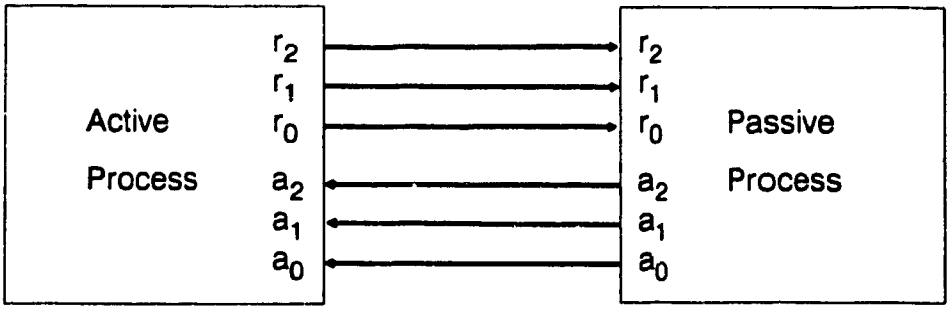
Figure 7-2 shows a possible state-transition table for the passive process of Figure 7-1(c). The acknowledge output A is dependent on both the input request R and the present state S All input channels can be characterized by such a Mealy machine table. The simplest case of a 2-wire channel would have only one input $R = \{r_0\}$, one output $A = \{a_0\}$, and one identical present and next state $S = \{s_0\}$. If that input channel changes a state variable in the process, then more states would be added. Although the change of state would not affect that input channel, it would affect other channels of the process.




(a) write data channel.




(b) read data channel.




(c) bidirectional data channel.

**Figure 7-1: Write, read, and bidirectional data channels.**

| input R | present state $S_n$ | output A | next state $S_{n+1}$ |
|---|---|---|---|
| $r_2$ | $s_0$ | $a_0$ | $s_1$ |
| $r_2$ | $s_1$ | $a_0$ | $s_1$ |
| $r_1$ | $s_0$ | $a_2$ | $s_0$ |
| $r_1$ | $s_1$ | $a_1$ | $s_0$ |
| $r_0$ | $s_0$ | $a_1$ | $s_1$ |
| $r_0$ | $s_1$ | $a_2$ | $s_1$ |

**Figure 7-2: A possible state-transition table for a passive process communicating with a bidirectional data channel, $R=\{r_0, r_1, r_2\}$, $A=\{a_0, a_1, a_2\}$, and $S=\{s_0, s_1\}$.**

Bidirectional channels may be used for general communication between processes. In the case of accessing memory modules, the requests can be used for writing the address of the data required and the acknowledges can return the data from that address. This is as opposed to using a separate write data channel to write the address and a read data channel to read the addressed data.

As a final generalization, a channel C can have w = b(an + cm) where w, b, n, and m are as before. The variables a and c indicate the number of request wires n and acknowledge wires m that occur in parallel in the bit channel. For example, if b = 2, a = 4, n = 3, c = 1, and m = 2, then w = 2(4n + m). This indicates two bit channels in parallel (b = 2), each bit channel consisting of 4 concurrent (a = 4) ternary bits (n = 3). The acknowledge for each bit channel is a single binary bit (m = 2).

## 7.2 Conversion of Channels

Different data channels can be converted to allow compatibility with the single channel DI building blocks. Figure 7-3 shows the conversion required for write data channels. The interface of process P consists of 2n signalling write data channels, an input (passive) write data channel on the left and an output (active) write data channel on the right. Conversion of the input write data channel between 2n signalling and n+1 signalling can be done simply by combining all the 2n channel acknowledge signals into one acknowledge signal with an OR gate. There is a one-to-one correspondence between the single request lines and the 2n signalling channels.

Conversion of the output write data channel will require C-elements. Request outputs from $W_2$, $W_1$, and $W_0$ flow through unchanged, but the single returning acknowledge, $a_0$, must know which channel to acknowledge. It cannot fork out and acknowledge every single channel, even the ones that did not make a request. The C-elements ensure that only the requesting channel gets acknowledged.

Conversion of read data channels is shown in Figure 7-4. The channel process P works as follows. To read from the process, a channel signal R comes in on the left and the resulting data is sent out from any one of the $A_2$, $A_1$, $A_0$ channels on the left. If the process P is the initiator, then it sends a channel signal R out on the right and receives the returned data on any one of the $A_2$, $A_1$, $A_0$ channels on the right. The conversion circuits is as shown. Note that the channel signals are shuffled in this implementation, so the process P must be careful in handling these channels.

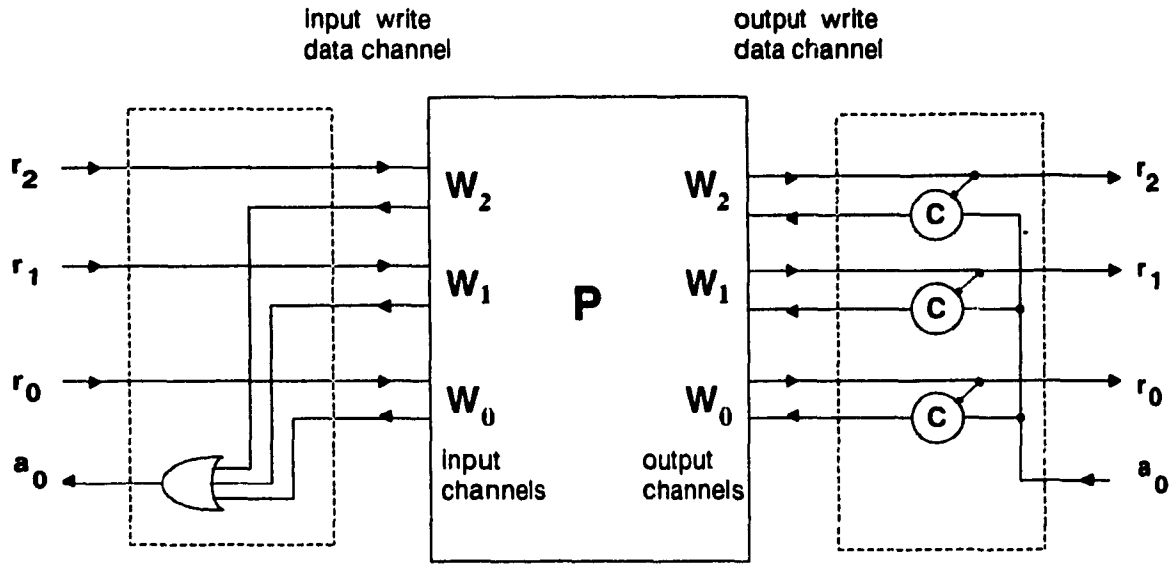



**Figure 7-3: Conversion of input and output write data channels (for b = 1, n = 3, m = 1).**

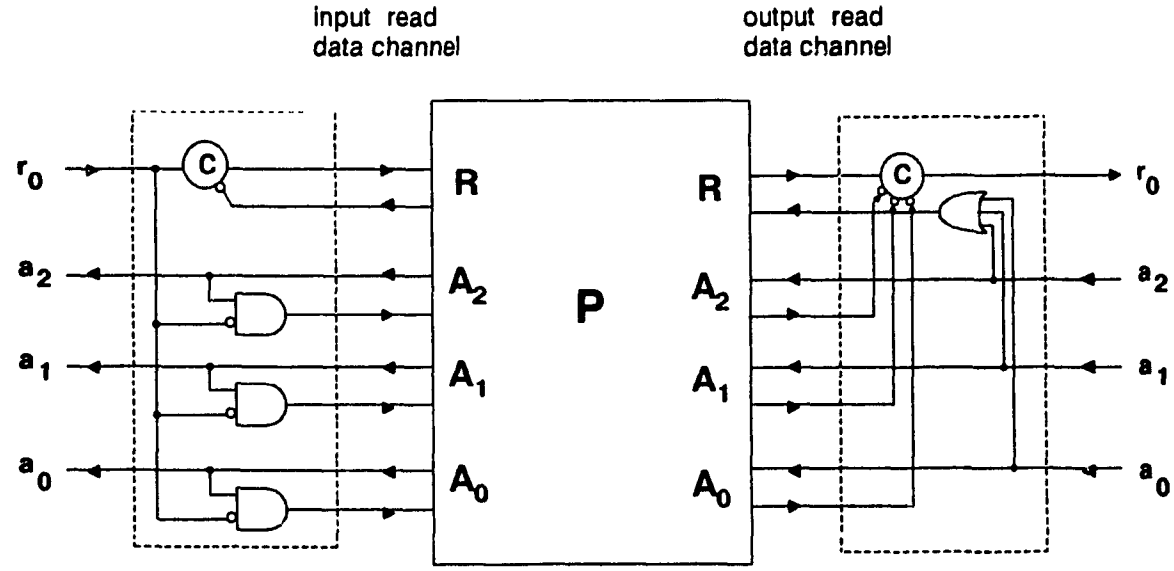



**Figure 7-4: Conversion of input and output read data channels (for b = 1, n = 1, m = 3).**

# Chapter 8

## Modes of Operation and the Correctness of Composition

There can be two modes of operation for the DI and hybrid building blocks. One is the normal delay-insensitive channel mode and the other is the delay-sensitive pulse mode. Here, we expand on how the pulse mode can be used, with regard to its better area-time performance. As well, the correctness of the system must be considered when it is composed out of the DI building blocks. To form a higher level system, two properties must be considered in order to ensure correctness, namely the properties of safety and liveness.

### 8.1 Modes of Operation

There are two modes of operation in which the DI and hybrid building blocks can operate on. They are:

1) channel mode (inter-module)
2) pulse mode (intra-module)

Channel mode is the basic mode of operation for the DI building blocks. Communication occurs only over delay-insensitive channels. Only modules which have solely channels as interfaces (DI modules) are allowed to operate in this mode. Since channels satisfy all the properties of delay-insensitivity, composition of DI modules is simply the connection of active and passive channels together, regardless of the delays of the interconnecting wires or the delays of the modules. In the channel mode of operation, no delay analysis is required. Communication is abstracted into the discrete packet world.

The channel mode is the primary mode of operation for the delay-insensitive circuits. The pulse mode is used only at the lowest level; it occurs only inside of the DI modules (intra-module). An example of this mode of operation was in the route-through state variable and the single-wire P/A-blocks. The pulse mode is

not necessary in the construction of DI systems. Indeed, we can consider the DI building blocks as the atomic blocks and need not go deeper into the lower level details. The purpose of considering pulse mode here is to clarify how the hybrid building blocks work and to describe their delay requirements. As well, pulse mode is considered so that there can be an alternative, more area efficient method of constructing a system.

The pulse mode is used as a simple method of interfacing between the delay-insensitive channel world and the delay-sensitive single wire world. To obtain a pulse from a channel from inside of a module, either the request or acknowledge wire of the channel can be *tapped*. The result seen from this tapped point is a pulse because the channel wires must go high and then low whenever a single 4-phase handshake sequence takes place. Figure 8-1 illustrates this (the example shown occurs in the route-through state variable). There exists a route-through channel, whose passive input is P and active output is A. Q1 and Q2 represent the tapped points where a pulse can be obtained from the request or acknowledge lines of the channel. The width of the pulse obtained will depend upon the speed with which the handshake sequence occurs. A faster handshake sequence will result in a narrower pulse and a slower handshake sequence will result in a wider pulse. Depending upon the pulse mode circuit

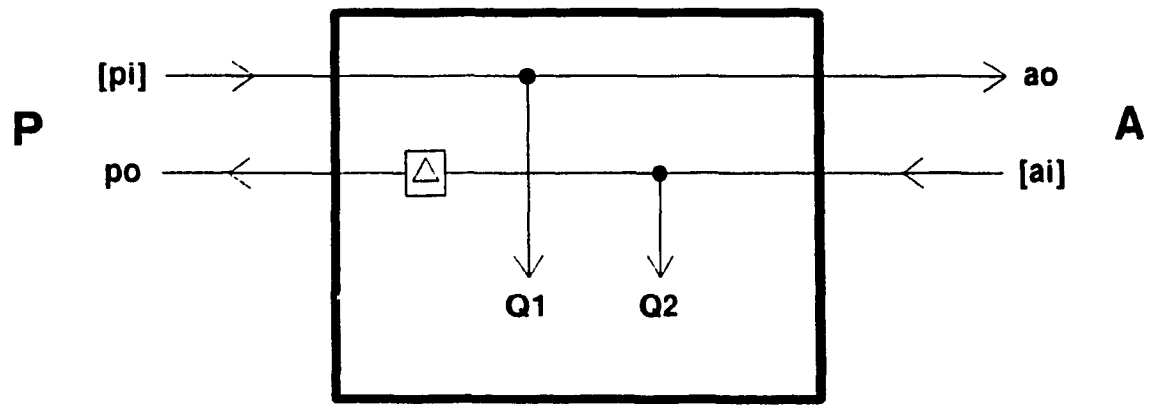



**Figure 8-1: Tapping of a channel to obtain a pulse in pulse mode of operation.**

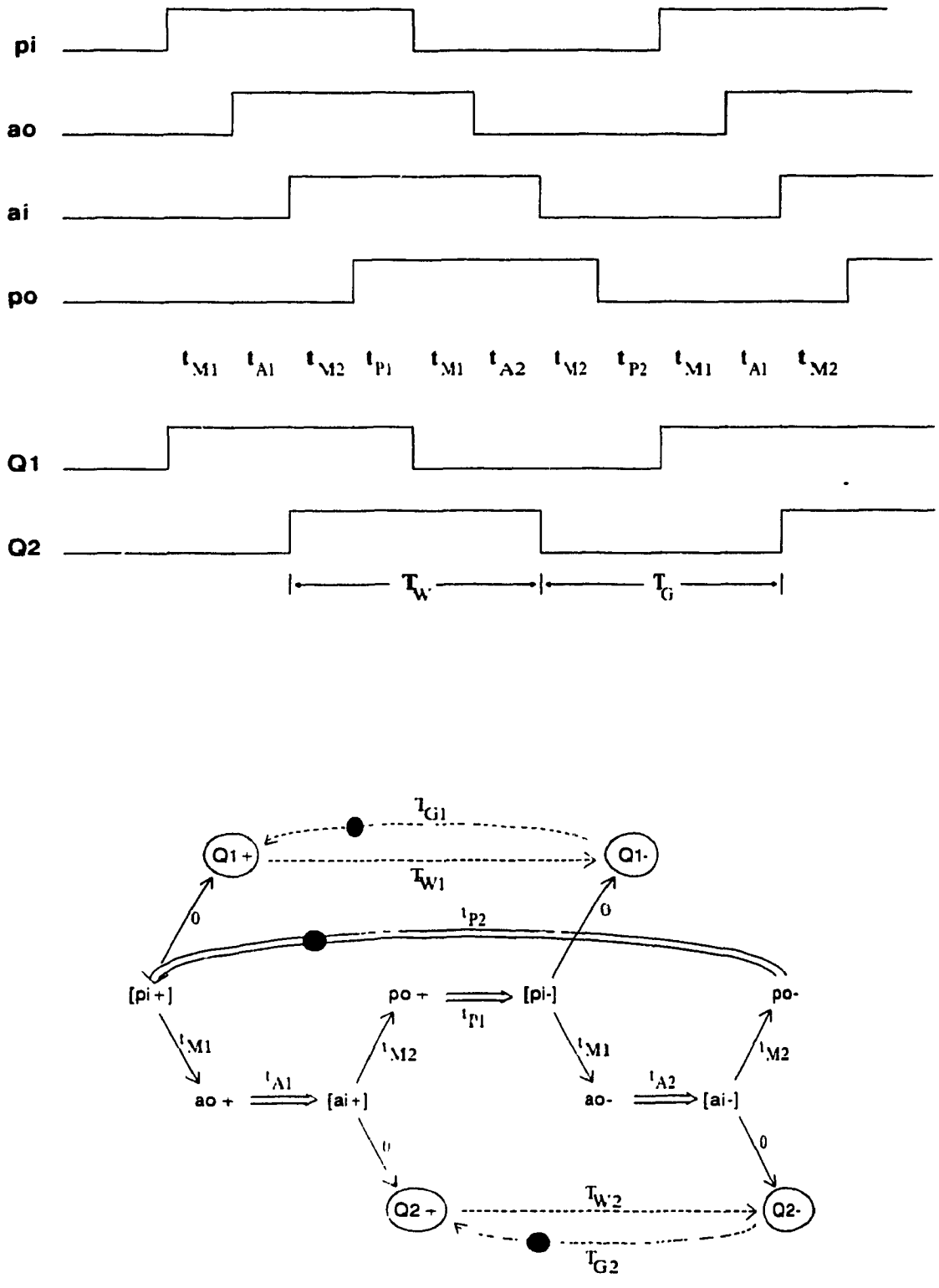



**Figure 8-2: Timing diagram and extended STG of previous pulse mode connection.**

inside the module, it may be required that the pulse be of a sufficient width to ensure correct operation. This is analogous to a system clock in which the width of the clock pulse must be of a sufficient duration to allow for all operations to occur within that duration (clocked systems are also considered as an example of pulse mode of operation).

To clarify the delays involved, Figure 8-2 shows the timing analysis in terms of a timing diagram and the extended STG model (timing diagrams usually use arrows to indicate dependencies between transitions, but this will become messy if many concurrent events occur). We are interested in the width of the pulse obtained from Q1 and Q2. The time $T_W$ indicates the width of the pulse obtained and the time $T_G$ indicates the gap between the pulses. $T_W$ and $T_G$ are dependent on the following delays:

$t_{M1}$ : module delay from the request wire
$t_{M2}$ : module delay from the acknowledge wire
$t_{P1}$ : environment delay from channel P (high to low transition)
$t_{P2}$ : environment delay from channel P (low to high transition)
$t_{A1}$ : environment delay from channel A (high to high transition)
$t_{A2}$ : environment delay from channel A (low to low transition)

For Q1, we have

$$T_{W1} = t_{M1} + t_{A1} + t_{M2} + t_{P1}$$

$$T_{G1} = t_{M1} + t_{A2} + t_{M2} + t_{P2}$$

and similarly for Q2. To alter the width of the pulse, we can control the module delays $t_{M1}$ and $t_{M2}$. To obtain a pulse of sufficient width to activate the pulse mode circuit, a delay element can be inserted to increase $t_{M1}$ or $t_{M2}$. Note that the insertion of a delay element actually increases the width of the pulse and not just simply delays the occurrence of the transition.

The resulting pulse can then be used in pulse mode to

1) set/reset a state variable, as a clock pulse to increment a counter, etc., and
2) sequencing.

In the first case, the pulse can be used for either positive or negative edge triggering of events, just as in edge sensitive clocked circuits. Depending on the location of the tapped point and placement of the delay element, different results can be obtained. The recommended location of the tapped point is at Q1. This allows the pulse circuit to react immediately upon an input request on the channel, thus allowing the most time for the pulsed circuit to become stable. The requirement is that the circuit becomes stable before the final low acknowledge is returned at po. A delay element can be inserted anywhere after the tapped point to allow for a sufficient $T_W$ and $T_G$ time. Note that for efficiency, the environment delays can be considered in the pulse width times, and so the necessity for a delay element may be eliminated.

The second case use of the pulse is for sequencing. This is a special form of pulse mode in which (i) the acknowledge wire is tapped, and (ii) the negative edge of the resulting pulse is used to initiate the next operation. This is the case where Q2 is used. The sequencing is between channel A and the pulse mode circuit. The operations on channel A is said to occur first because the handshake sequence on A is completely finished before the returning low acknowledge on [ai] comes in and initiates the pulse mode circuit. The circuit is negative edge triggered. An example of this sequencing is in the single wire P/A-blocks. When the handshake sequence of a block is finished, its final low acknowledge initiates the next block. Another example is in the test-and-set and alternator blocks. A channel signal must route through the block first and then set the state variable upon completing the channel signal. Subsequent channel signals would then be affected by the set state variable. To ensure sufficient time for the negative edge triggering to affect the circuit, a delay element can be placed after the Q2 point.

This type of sequencing is very useful whenever a channel signal alters itself. Data can flow through a channel from the source to the destination. Once that data has completely been sent, the path for the next data is changed by changing the controlling state variable. The present data alters the control flow for the next data.

The use of this pulse mode of operation for sequencing can reduce both the area and time performance of the system. In order to perform this type of sequencing operation, for example in the test-and-set, a second channel would have been required. One channel would be used to make the test as before, and an additional second channel required to set the state variable. The increased area cost is in the additional channel and its control circuitry. The increased time cost is in the time taken for the execution of the second channel and in the strict serialization of the first channel and the setting of the state variable. The use of pulse mode allows for the channel signal and the setting of the state variable to be interleaved, allowing the output of the state variable to change as soon as the channel signal has completed.

The use of pulse mode also allows for other possibilities of trading off delay-insensitivity for increased area-time efficiency. Synchronous systems are essentially a form of pulse mode circuits, the clock pulse being used to fork out and initiate all operations. Two-phase clocks rely on two pulses instead of one. Delay analysis is required to ensure that the clock pulse width is long enough to allow for all operations to occur. To interface delay-insensitive channels to synchronous (locally clocked) systems, a channel can be used to obtain a pulse. That pulse can then be used to initiate a series of pulses, or used as is for controlling data latches. There is the option of using the DI building blocks as asynchronous control flow circuits to control latched bundled data wires, as opposed to using them in only totally delay-insensitive systems. The use of structured building blocks would produce much clearer self-timed circuits than that of the presently existing ad hoc gate level designs.

## 8.2 Correctness of Composition

As individual building blocks are composed together to form higher level systems, there arises the concern for the correctness of the system as a result of the composition. When the DI building blocks are connected together to form a network, it must be ensured that the resulting network operates properly. That is, deadlock should never occur in the resulting system, given a certain input, the correct output must be returned, etc. In chapter 4, correctness was discussed in terms of the hierarchical composition of STG specifications. To verify correctness, the STG specifications of individual modules are composed together to obtain a composite specification, and then the result is compared with the desired behavior. Composition is used at each level of the hierarchy, from the bottom up, to ensure that the entire system is correct. There has been much work done on the correctness of concurrent systems, namely in the area of concurrent programming. To prove that a program is correct, one of the known methods is to show that a program satisfies the properties of safety and liveness [Owic82, Prob88]. Concurrent hardware systems are closely related to concurrent software systems. In this section, we show how the correctness of DI circuits are related to the correctness of software systems. The intention is not to delve deeply into proof of correctness issues, but to provide a general feeling of what it means to have a correct system implemented out of the DI building blocks.

In software systems there are two properties that a concurrent program should satisfy to ensure correctness: safety and liveness. The safety property requires that the program never enters an unacceptable state (something bad never happens) and the liveness property requires that the program eventually enters a desirable state (something good eventually happens). Examples of safety are absence of deadlock in a program and mutual exclusion of processes sharing the same resource. In terms of modules/environments, safety requires that the environment send in only correct inputs that the module expects

(meaning no computation interference). For example, if the module expects mutually exclusive data from two input ports, the environment cannot send in two concurrent data values. If the module expects input data to occur only in a certain sequence, then the environment cannot send in the data in an out of order sequence.

In the case of liveness, a certain event must eventually occur given some initial condition. One instance of liveness is that a program must eventually terminate. Another is that a process sharing a resource must not be allowed to starve. If message based communication is used, then the message must eventually reach its destination. In terms of DI modules, a message will always reach its destination because request/acknowledge channels are used (no transmission interference). Since our delay-insensitive circuits always use channels, reliable communication is assumed, and so this aspect of liveness is not a concern. Liveness also requires that a module eventually emit an output if an input is given. For example, given a FIFO queue of size N, N data values must eventually be output if there were originally N input data values (safety will require that the correct ordering of the output data values be retained). Safety and liveness are closely related and are complementary. To prove liveness, one usually has to prove a number of safety conditions.

In correctness through the composition of STG specifications, safety and liveness can also be ensured. Safety corresponds to environment constraints and liveness corresponds to the liveness of the module. When STG's are composed together, the environment constraints (safety conditions) of each individual STG must be ensured. For example, if the specification of a module M requires that certain inputs occur mutually exclusively (indicated by a shared place sitting on environment double arrows), then a composition involving the STG of M must ensure that this condition is not violated. Since the input environment of M is now the output of the other modules in the composed network, it must be ensured that the other modules only send M mutually exclusive signals.

To ensure liveness in an STG composition, the resulting STG of the composition must be live. That is, the resulting STG must be live in the same sense as Petri nets being live. Analysis must be done on the STG/Petri net to assure freedom from deadlock, and the firing behavior of the STG must also be cyclic. When the environment safety conditions of each individual STG are satisfied, the composite STG will be live. At each level of the hierarchy, once the safety conditions of the network of modules are satisfied, the composite higher level module will be live and can then be used in the next higher level of composition. At the lowest level of the hierarchy, the atomic DI building blocks are assumed to be live.

Note that the Petri net analysis required at each level to assure freedom from deadlock may not lead to exponential analysis time for the entire system. At each level of the hierarchy, the composed STG becomes smaller (contracts). Exponential analysis time would occur only if the composition of the system is done all at a single level (eg. all the STG's of the lowest level DI building blocks are composed and analyzed at a single level) rather than hierarchically.

As an example to illustrate how the correctness of the system is affected by the interconnection of individual modules, Figure 8-3 shows a system composed of three processes, A, B, and C. Each process is a DI module communicating with channels. Process A first sends a packet to process B (with a quick return interface) which absorbs the packet and then sends a packet to process C. Process A, having finished sending a packet to process B, will then send a packet to process C also. The two resulting packets shown in the figure will have a race to process C. There is no problem if process C synchronizes the two packets to come in together concurrently with a join building block. But in some cases, the two packets at the input of process C must remain separated, and C must receive one packet before the other. For example, this can occur if the top input channel of C sets a state variable which will control the output route of the packet entering the bottom input channel The order in which these two packets arrive will affect the output of process C.

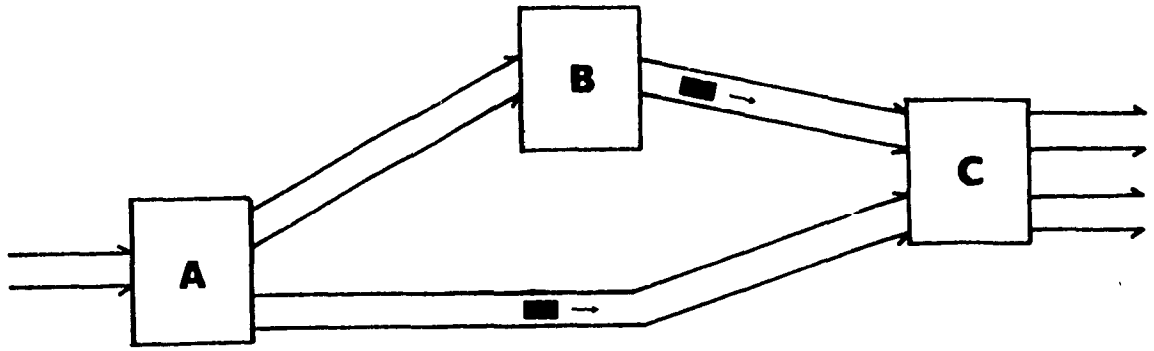



**Figure 8-3: Race condition at the channel level.**

If this is the case, then the environment specification of process C (mutual exclusion of its two input channels) has been violated as a result of an incorrect composition. This particular interconnection with process A and B would incorrectly result in concurrent inputs to C. The error occurs from having a quick-return interface at process B which gives a premature acknowledge signal to process A, allowing A to proceed and send a concurrent packet to C. The problem can be solved by having a route-through channel in process B so that the initial packet routes through B and is absorbed by C before A sends the second packet to C. An alternative method is to include an additional "acknowledge" channel from process C to A so that the initial packet is acknowledged by C through this channel before the second packet is sent by A.

If provisions are not made to explicitly order the input packets to process C, then non-determinism will occur. The output of process C would be non-deterministic because the order of occurrence of the input packets to C are not known as a result of the delays of the packets on the input channels. It is interesting to see that even the use of channels cannot avoid delay-sensitive races. Although the individual channels by themselves are delay-insensitive, its

use in a higher level composition may be delay-sensitive. Unlike asynchronous circuit design at the wire level, this race problem is not solved by the manipulation of delays, but is solved by avoiding this composition.

This type of packet race also occurs in message passing distributed computer systems. Unless the packets are explicitly forced to arrive in a certain order or are synchronized to come in concurrently, non-determinism will result. This type of non-determinism is slightly different from the usual non-determinism as a result of arbiters. It is caused by the differences in the delays of the message paths, rather than the explicit arbitration hardware. Like the example shown, a message may also incur delays through intermediate processes before reaching its destination. Delay-sensitive non-determinism may or may not be desired, depending upon the specification of the system. Delay-sensitive non-determinism is usually not fair because the physical message links or wires in a circuit usually have different lengths, depending upon the layout, and are fixed when the system is constructed.

# Chapter 9

# Example Designs

To demonstrate the use of the DI building blocks, two circuit designs will be shown: a DI finite state machine (FSM) and a circuit for distributed mutual exclusion (DME). The FSM is implemented using fine-grain DI modules which are composed hierarchically, as opposed to implementing the FSM at a single "flat" gate level. Although this particular implementation is not area efficient, we use it primarily to demonstrate how most of the DI building blocks can be used. The second example circuit design, the DME, is a circuit for implementing mutual exclusion among a number of users. Since there exists a previous implementation of such a circuit using a speed-independent design technique, some comparisons can be made. Language notations for the circuits implemented out of the DI building blocks will also be discussed.

## 9.1 DI Finite State Machine

The type of finite state machine that we want to build is a Mealy machine, an FSM with output. For implementation, we arbitrarily choose an FSM that recognizes the bit pattern "101". Whenever it recognizes this bit pattern in a string of input bits, it emits a "1", otherwise it emits a "0". The implementation of the FSM using the DI building blocks must be able to represent (1) the states, (2) the input/output, and (3) the transition from state to state.

The initial design of the FSM and its Mealy machine state diagram is shown in Figure 9-1. Three states are required, namely A, B, and C. The circuit implementation uses a separate DI module called a State block to represent each state. To record the present state of the FSM, each State block contains a "present_state" variable which can take on the value "1" or "0". Only the present state State block contains the value "1", while all the other State blocks contain

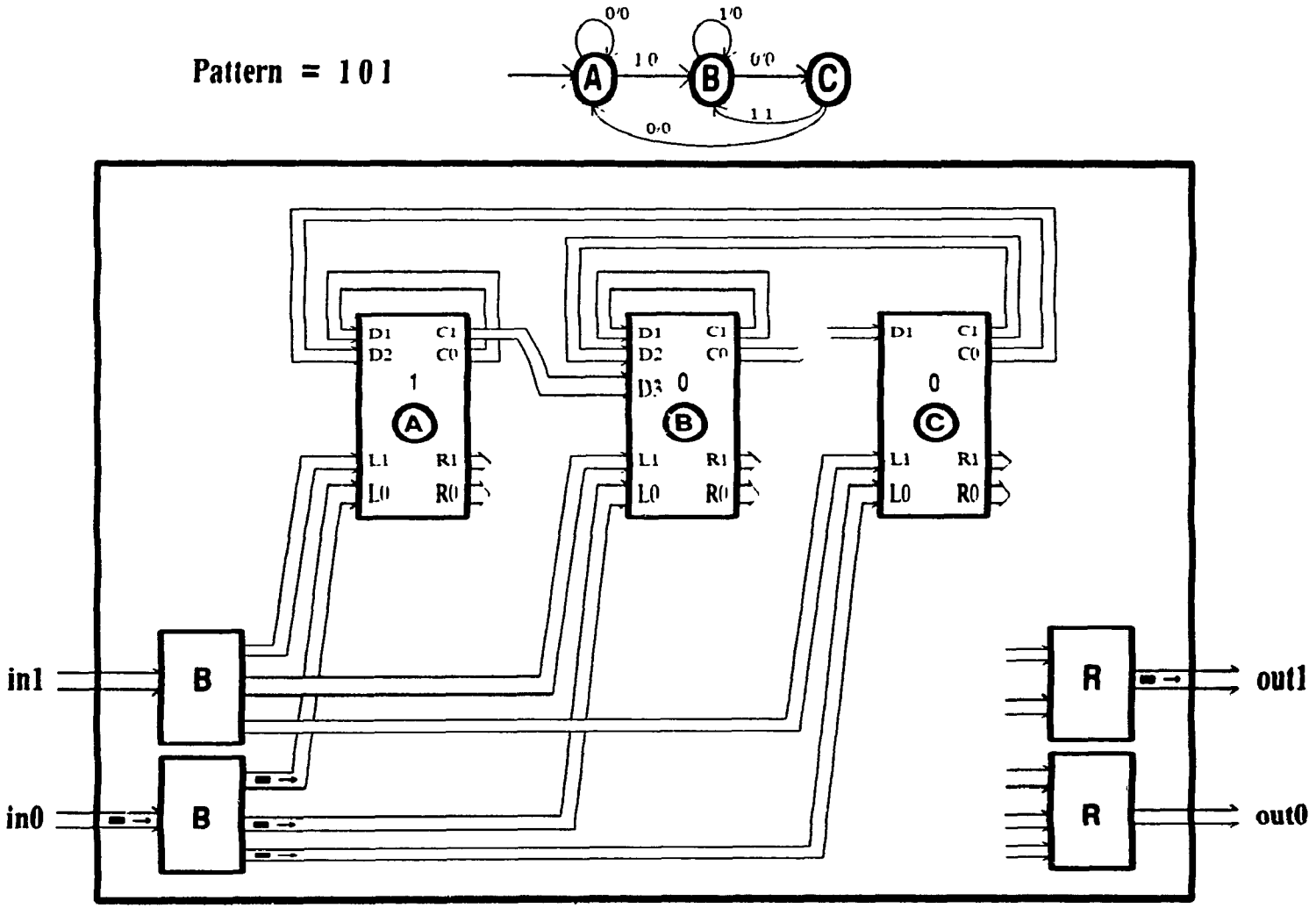



**Figure 9-1: DI Finite State Machine.**

the value "0". The start state of the FSM is set by initializing "present_state" = 1 in State block A at system power on.

The input alphabet of this FSM is $A_{IN} = \{1, 0\}$ and the output alphabet is $A_{OUT} = \{1, 0\}$. This is easily encoded using channels, one channel representing a "1" and another channel representing a "0". The interface for the DI FSM consists of an input bit channel {in1, in0} and an output bit channel {out1, out0}.

The operation of the FSM must first accept an input value on the input bit channel, emit an output value on the output bit channel, and then make a transition to the next state depending upon the input value. The input of the FSM goes through Broadcast blocks so that the input data value information can be sent to all the State blocks. Since the input alphabet is {1, 0}, two Broadcast blocks are required, one for each bit value. Upon receiving an input packet, a "0" for instance as shown in the figure, the lower Broadcast block will broadcast a "0" packet, called a broadcast packet, to all the State blocks. The broadcast is done concurrently to all the states. If the State block has a "present_state" = 1, then the broadcast packet will route-through the State block from L0 to R0 and be sent to one of the Reception blocks. The Reception blocks are the complement of the Broadcast blocks and are used to collect packets. Whenever a packet comes into any one of its input channels, it gets sent to the FSM output. Only the single State block which has "present_state" = 1 can route its broadcast packet to the Reception block. All the other State blocks with "present_state" = 0 simply absorb the broadcast packet that is sent to it. In this way, only the present state State block controls the output value emitted by the FSM. Programming the output data value to be emitted is simply the connection of the R1/R0 output channel of the present state State block to either the "1" or "0" Reception block. This implements the Mealy machine requirement of accepting an input value and emitting an output value depending on both the input value and the present state.

To implement the transition from the present state to the next state, the single State block with "present_state" = 1 must clear its own state variable to "0" and set the "present_state" variable of the next State block to a "1". This is done

through the C-D channels. Once the State block has sent its output data packet at its R1/R0 channel, it will clear its own "present_state" variable and send a packet, called a transition packet, on its C1/C0 channel to the next State block. If the input to the FSM was a "0" then the transition packet is sent out on its C0 channel, otherwise, it would be sent out on its C1 channel. This transition packet will then enter the D channel input of the next State block and tell that State block to become the new present state. After all the broadcast packets to the State blocks have been processed, the next input to the FSM can occur. The simulation of the next state transition is therefore done by passing a transition packet around on the C-D channels, one pass occurring on each input to the FSM. The "present_state" variable is a token which gets passed from one State block to the next.

Using this implementation of the FSM, there may exist the problem of packet races as discussed in Section 8.2. The delays with which a packet message is sent may affect the correctness of operation. This race may occur between a broadcast packet and the transition packet. The specification of the State block requires that inputs to its D channel and its L1/L0 channel be mutually exclusive. This is because broadcast packets that enter the L1/L0 channel must be evaluated by the "present_state" variable, whether to simply absorb the broadcast packet or to route it through to the Reception block. But at the same time, the transition packet may be entering the D channel to change the "present_state" variable. The order with which the broadcast packet and the transition packet come into the State block will cause different effects. The desired behavior is that the broadcast packet win the race (that all the broadcast packets have approximately equal delay paths) because the transition packet is meant to control only the next input (next broadcast) in the FSM. This occurs most of the time because the transition packet incurs a larger delay; upon receiving its broadcast packet, the present State block must first send the packet to the Reception block before sending its transition packet.

But to guarantee total delay-insensitivity, some changes must be made. The problem can be solved by separating the sending of the transition packet from the initial broadcast. Two broadcasts can be used, the first one used to tell the present State block to send its output to the Reception block and the second broadcast to tell the present State block to send the transition packet to the next State block.

Note that broadcasts are required because the input packet to the FSM does not know which of the State blocks is the present state. The input packet cannot selectively be sent to the correct state because the state information is distributed throughout the DI system in the State blocks. Alternatively, finding the present state can be done nearly from one State block to the next rather than concurrently with a broadcast. This would give an average case evaluation time of O(N) rather than the O(log N) time used in the broadcast method, where N is the number of State blocks.

The actual implementations and corresponding specifications of the Broadcast, Reception, and State blocks of the FSM are shown in Figure 9-2 to 9-7. The Broadcast block consists of a P/A-block sequence and two fork building blocks. The fork blocks are used to implement the two broadcasts, the first broadcast being sent on the channels {R1. R2, ... Rn}, where n is the number of State blocks required in the FSM; the second broadcast which is used to initiate the next state transition is sent on the channels {N1, N2, ... Nn}. The P/A-blocks are used to serialize the two broadcasts. Upon an input packet to the P-block, the two A-blocks will output a packet which will be forked out into broadcast packets.

The Reception block is implemented similarly to the Broadcast block. It contains an mjoin which collects input packets coming into any one of its input channels and sends it to the output. Note that the mjoin specification requires mutually exclusive input packets. This is ensured by the interconnection of the FSM because there exists only one mutually exclusive present state; and it is responsible for sending the mjoin the single packet. The P/A-block sequence

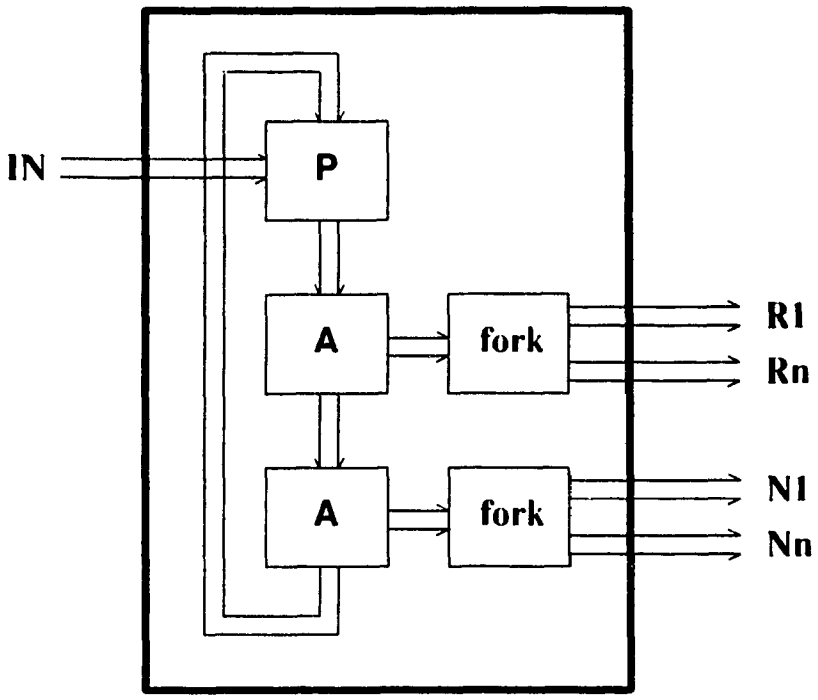



**Figure 9-2: Broadcast block.**

4-phase

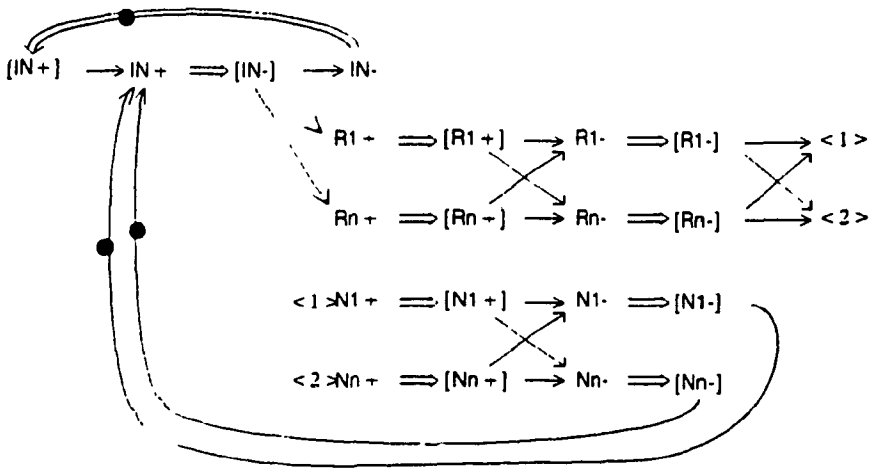



**Figure 9-3: Specification of Broadcast block.**

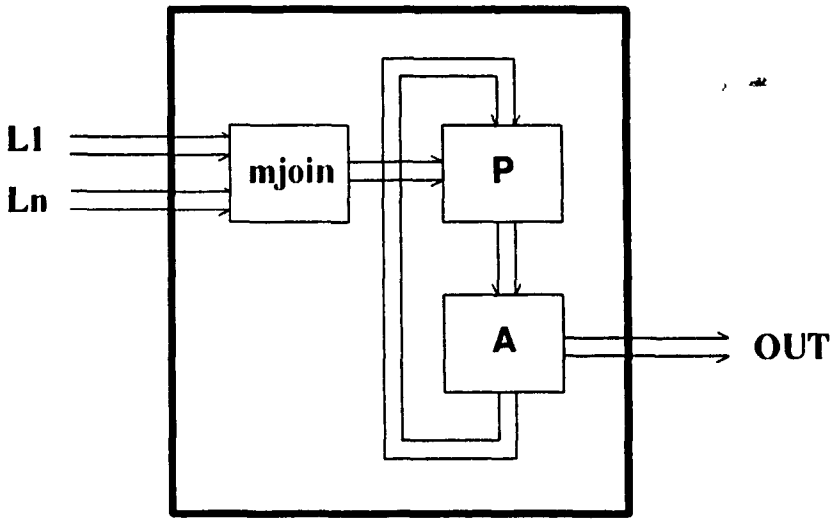



Figure 9-4: Reception block.

4-phase

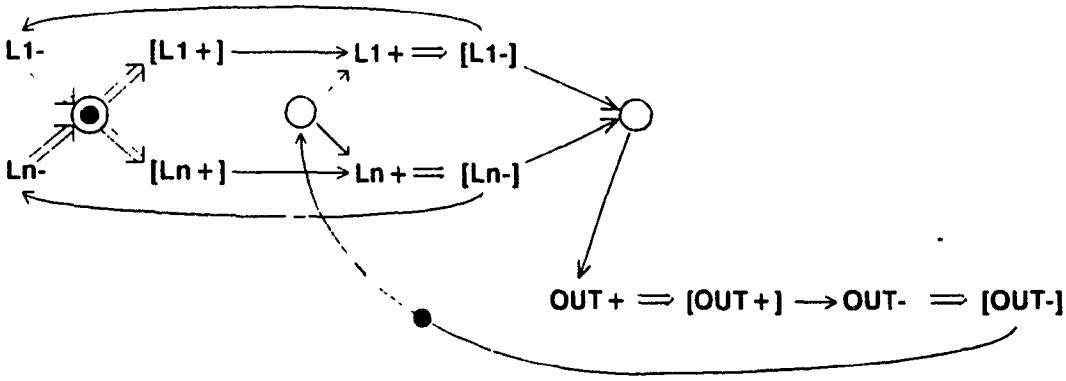



Figure 9-5: Specification of Reception block.

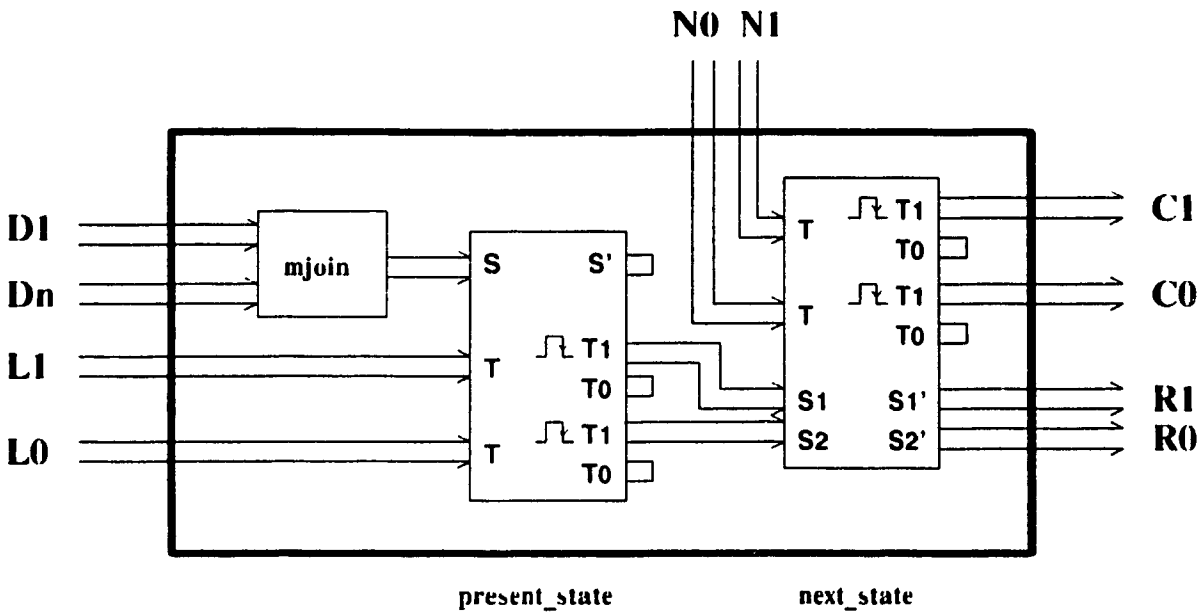



**Figure 9-6: State block.**

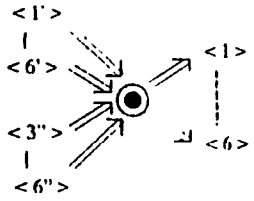
consists of only two blocks, which is equivalent to a simple FIFO element. Although this is not required its inclusion allows the FSM to work in a pipelined fashion. Note that since the P-blocks are quick-return blocks, the "1" and "0" Reception blocks must be made to operate mutually exclusively. Otherwise, the "1" and "0" packets may overtake each other and result in an incorrect output. The connection shown previously in Figure 6-24 is required to maintain mutual exclusion of the two Reception blocks. This similarly applies to the two Broadcast blocks.

The State block implementation in Figure 9-6 consists of two test-and-reset blocks and one mjoin block. The test-and-reset blocks contain simple extensions to the basic test-and-reset and can be easily implemented. The first test-and-reset records the "present_state" variable. If it is set to "1", then the

ps = present_state
ns = next_state

ps = 1, ns = 0 : State block is present state (initially).
ps = 0, ns = 0 : State block is not present state.

4-phase




ps: = 1
<1> ⟹ [D1+] ⟶ D1+ ⟹ [D1-] ⟶ D1- ⟹ <1'>

ps: = 1
<2> ⟹ [Dn+] ⟶ Dn+ ⟹ [Dn-] ⟶ Dn- ⟹ <2'>

<3> ⟹ [L1+] —ps=1, ns=1→ R1+ ⟹ [R1+] ⟶ L1+ ⟹ [L1-] ⟶ R1- ⟹ [R1-] ⟶ L1- ⟹ <3'>
ps: = 0
[L1+] —ps=0→ L1+ ⟹ [L1-] ⟶ L1- ⟹ <3">

<4> ⟹ [L0+] —ps=1, ns=1→ R0+ ⟹ [R0+] ⟶ L0+ ⟹ [L0-] ⟶ R0- ⟹ [R0-] ⟶ L0- ⟹ <4'>
ps: = 0
[L0+] —ps=0→ L0+ ⟹ [L0-] ⟶ L0- ⟹ <4">

<5> ⟹ [N1+] —ns=1→ C1+ ⟹ [C1+] ⟶ N1+ ⟹ [N1-] ⟶ C1- ⟹ [C1-] ⟶ N1- ⟹ <5'>
ns: = 0
[N1+] —ns=0→ N1+ ⟹ [N1-] ⟶ N1- ⟹ <5">

<6> ⟹ [N0+] —ns=1→ C0+ ⟹ [C0+] ⟶ N0+ ⟹ [N0-] ⟶ C0- ⟹ [C0-] ⟶ N0- ⟹ <6'>
ns: = 0
[N0+] —ns=0→ N0+ ⟹ [N0-] ⟶ N0- ⟹ <6">

**Figure 9-7: Specification of State block.**

State block is the present state, and if it is reset to "0", then it is not the present state. The second test-and-reset block is required for storing a "next_state" variable. This second variable is required because of the use of a second broadcast. It is initially set to "0" in all the blocks.

If the State block is the present state, then the first broadcast packet will enter the L channel of the State block. Since "present_state" = 1, the packet will be routed through the first test-and-reset to the T1 output channel. The packet then routes through the second test-and-reset and sets "next_state" = 1. The output packet on the R channel is sent to the Reception block. On the second broadcast, the broadcast packet enters the N channel this time. Since it sees "next_state" = 1, it will route the packet out to the C channel. This becomes the transition packet and is used to initialize the next State block, sending it the present state token. The transition packet enters the next State block through the D channel into the mjoin block, making its "present_state" = 1.

If the State block is not the present state, then "present_state" = 0 and "next_state" = 0. The first broadcast packet entering the L channel simply gets absorbed because the first test-and-reset block selects the T0 output channel to implement a simple quick-return. The second broadcast enters the N channel, and again, the packet is simply absorbed. The two broadcasts have no effect on the State blocks which are not the present state. A lower level implementation of the State block using hybrid building blocks is shown in Figure 9-8.

It is easy to extend the FSM by increasing the number of states or by increasing the input/output alphabet. For example, an input alphabet of $A_{IN} = \{1, 0, EOS\}$ allows for an end of string symbol. This can be used to signify the end of the binary bit string and to reset the FSM back to its initial start state. The output alphabet can also include the null symbol, $A_{OUT} = \{1, 0, NULL\}$. A null output can be sent by simply not sending the packet to the Reception block (implemented with a shorted wire quick-return at the R output channel of the State block). As well ternary bit strings can be used by adding an additional channel. Note that this FSM is a fairly large circuit. If it had been expanded into

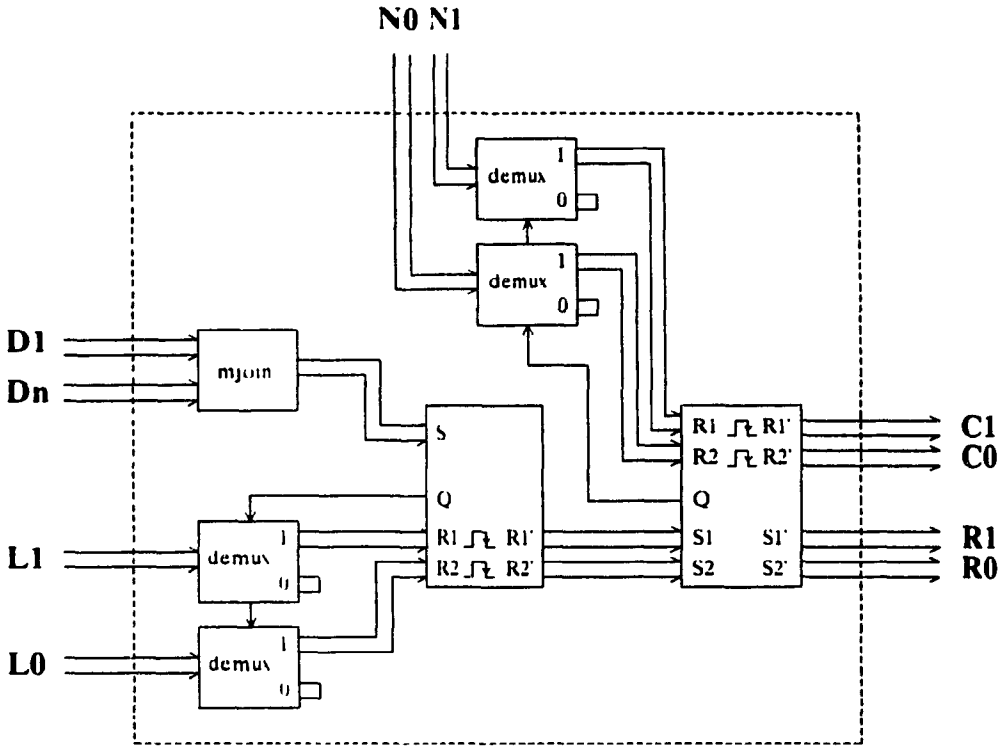



**Figure 9-8: Lower level implementation of State block using hybrid building blocks.**

a single flat gate level circuit, it would have been nearly impossible to comprehend.

## 9.2 Distributed Mutual Exclusion

The FSM just described was not intended to be area efficient, but was meant to serve as a demonstration circuit employing most of the DI building blocks. In this second example, we choose a circuit which has already been implemented so that some "loose" comparisons can be made. The circuit is a distributed mutual exclusion (DME) element which was designed by Martin in [Mart85] and later corrected in [Dill88]. It is a speed-independent circuit which was derived by compiling a CSP based language into lower level operators. The DME circuit is used as a possible solution for solving the problem of mutual exclusion among a number of users.

The high level implementation is shown in Figure 9-9. To achieve mutual exclusion among N users, a corresponding number of DME cells are connected in a ring. The cells are connected left channel to right channel, and the users request service from the ring via the top U channels. Mutual exclusion is implemented by the use of a single token which may reside in any one of the cells. Only the cell which contains the token can honor the corresponding user's request. If a request is made to a cell which does not contain the token, the request is sent to the right neighbor and on down the line until the token is found. Once the token is found, it is moved to the new cell location where the original user request was made, and the user is acknowledged. Subsequent requests by this user will be immediately acknowledged if other users have not requested the token. The implementation is called distributed because the DME cells can be distributed throughout the system. (In a sense, the FSM also implements distributed mutual exclusion; there is only one active State block which contains the "present_state" token. But unlike the DME, the mutual exclusion is deterministic, determined by the input to the FSM.)

The lower level implementation of a single DME cell using DI building blocks is shown in Figure 9-10, and its corresponding specification is shown in Figure 9-11 (our specification is slightly different from that of Martin's DME). The

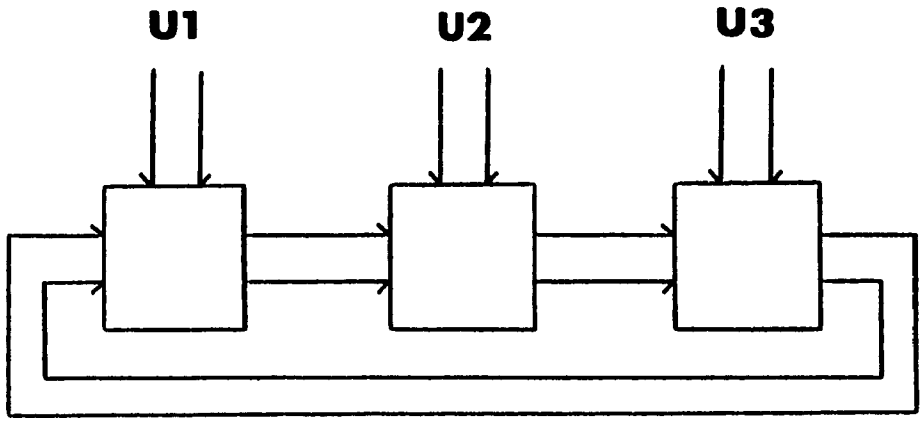



Figure 9-9: High level implementation of distributed mutual exclusion.

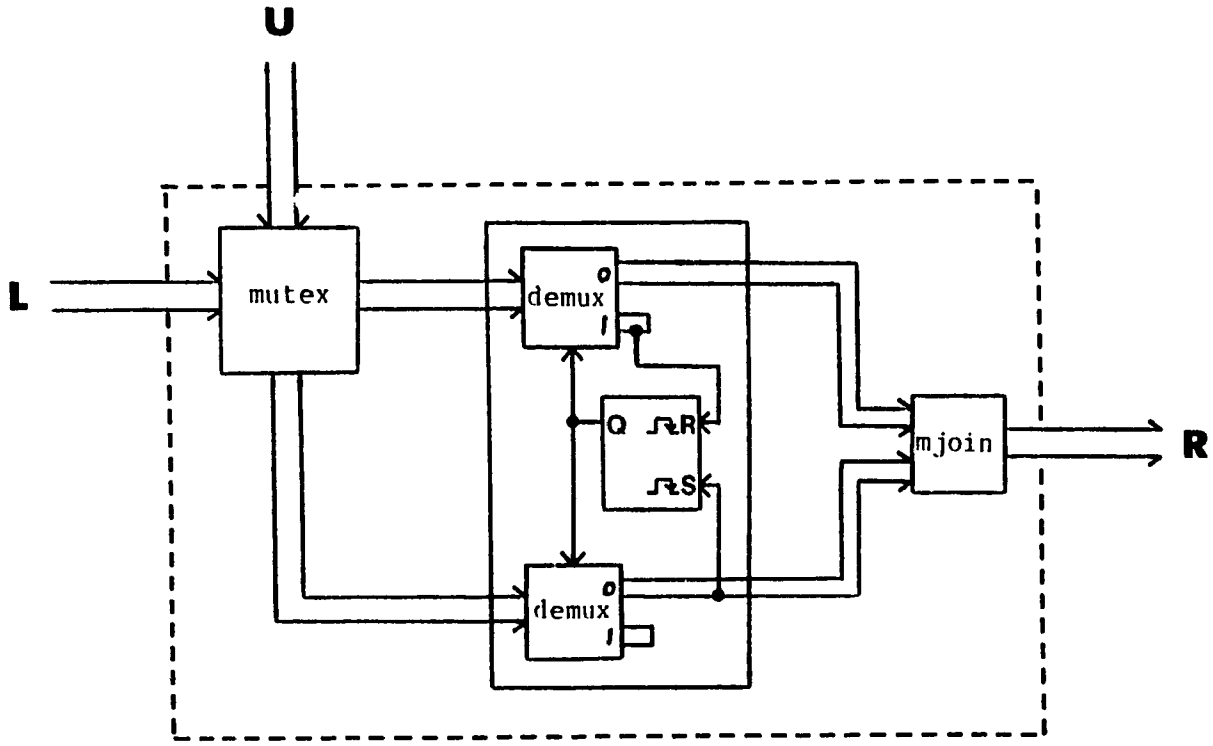



Figure 9-10: Lower level implementation of a single DME cell.

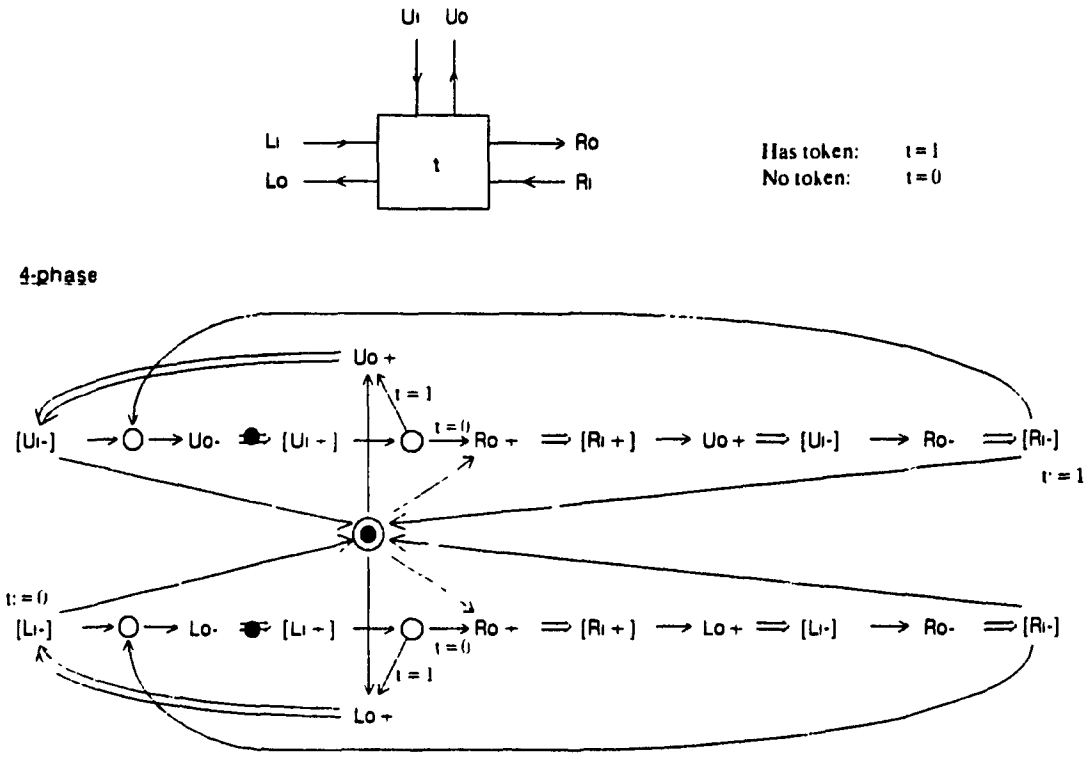



**Figure 9-11: Specification of the DME cell in Figure 9-10.**

DME cell consists of three DI modules, the middle one implemented with hybrid building blocks. The token is implemented as a state variable. If its Q output is a one, then the cell contains the token; if it is a zero, then the cell does not contain the token. User requests come in from the top U channel. The user request is arbitrated with the left neighbor's token request on the L channel. Assuming that the user wins the arbitration, its request gets routed down through the mutex block to the input of the lower demux block. If the cell contains the token, the state variable selects the lower "1" output channel of the demux, which is a shorted wire to implement a quick-return. The user's request therefore gets immediately acknowledged by the cell. If the cell does not contain the token, then the demux block routes the request through its "0" output channel and through the mjoin block to the R channel. The request is then sent to the cell's

right neighbor and on down the line until the token is found. The request coming into the right neighbor enters the DME cell's L channel, through the mutex block, and to the upper demux. Since the right neighbor has the token, the request goes through the "1" output of the upper demux and gets reflected back as an acknowledge. On the final low acknowledge of the 4-phase handshake sequence, it resets the state variable of the right neighbor and sets the state variable of the original requester, thereby moving the token to the new location. Note that all the blocks use route-through channels.

Our implementation can be compared with the corrected DME circuit implementation in [Dill88], although somewhat "loosely" because this is intended to be a delay-insensitive circuit (3 DI modules), while the latter is a speed-independent circuit (1 DI module). In terms of area, they are approximately equivalent (if inverters are ignored, our implementation uses one less gate). In terms of time, our implementation may be potentially faster. For example, if the cell contains the token, the request is almost immediately acknowledged and does not have to traverse through as many gates. The use of building blocks provides a much lower design time complexity and simplifies the understanding of the circuit.

Other implementations are also possible, depending upon the arrangement and granularity of the blocks. For example, the partially delay-sensitive hybrid building blocks can be replace by totally DI building blocks. Figure 9-12 shows possible replacements, either with two DI building blocks (a test-and-set and a test-and-reset) or with a dedicated general test block (a combined test-and-set/reset).

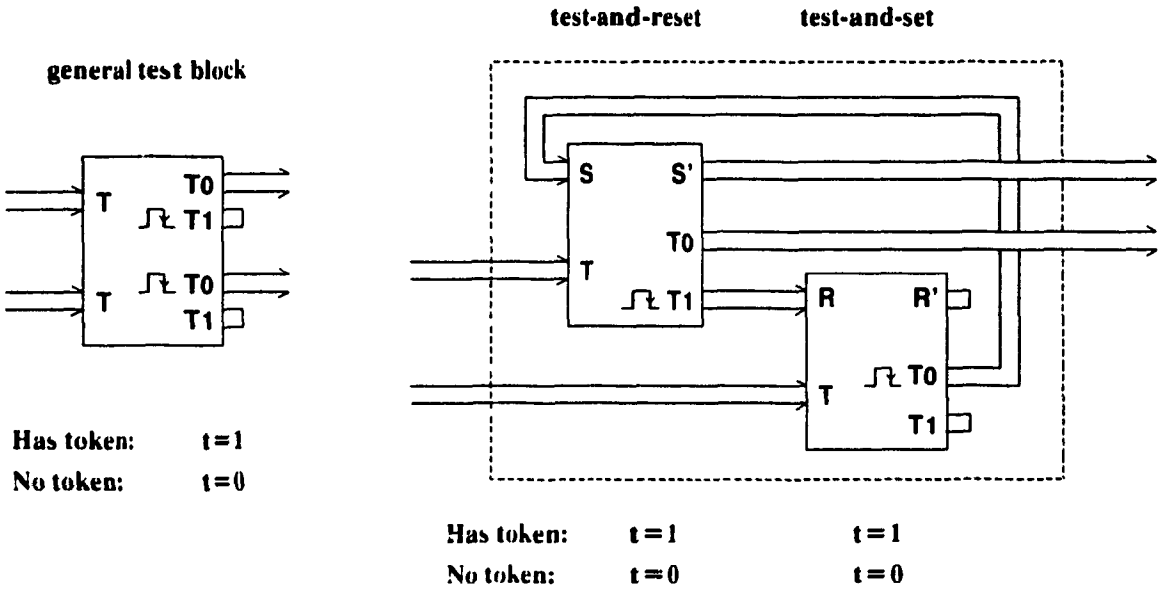



**Figure 9-12: Possible replacements of the delay-sensitive hybrid building blocks with totally DI building blocks.**

# Chapter 10

## Conclusion

A set of basic DI building blocks has been presented which can be used as the lowest level building blocks of a delay-insensitive silicon compiler. Extensions to the basic STG model were made so that the building blocks can be clearly specified and analyzed. A hierarchical composition procedure was outlined which may be used to verify the correctness of delay-insensitive circuits. A packet model was shown which allows VLSI circuits to be designed at a more abstract level. To demonstrate how the DI building blocks are used, two example circuits were shown: a DI finite state machine and a DME element.

Future work that remains to be done is to use the DI building blocks in an actual silicon compiler. A high level language is required for translating into the lower level DI building blocks. Possible candidates for implementation languages are CSP or functional languages such as Lisp. Alternatively, more specification oriented models can be used, such as Petri nets or data flow graphs. The universality of the DI building blocks should be proved, universal for the class of modules based on channel communication. One important aspect of a design is the verification of correctness. The procedure to compose STG specifications needs to be formally defined, and extensions to the composition procedure can be made to handle state variables, choice, and non-determinism.

The DI building blocks shown are at one extreme end of the spectrum, that of fine-grained delay-insensitivity. In general, finer grain DI modules require more area than larger grain DI modules; there is a tradeoff between area efficiency and greater degrees of delay-insensitivity. An alternative, more area efficient method than using purely DI modules is to use the DI building blocks as control circuits; data latches can interface to the channels via pulse mode,

thereby "piggy-backing" the data lines on top of the delay-insensitive control lines.

There are some interesting future effects for delay-insensitive circuits besides the advantages described in chapter 1. Issues relevant to synchronous circuits also apply to delay-insensitive circuits, such as fault-tolerance and testability. Testing synchronous circuits requires the forcing of all wires in the circuit to go high and low to make sure that there are no stuck-at faults. Much effort must go into forcing this transition and making it observable at the output. Since the request/acknowledge wires in a delay-insensitive circuit must go high and low anyway, the testing for faults may be simpler (all except for the internal state variables). Any stuck-at faults, single or multiple, would eventually be observed in the environment as stuck-at acknowledge lines (a dead system). As well, there are interesting effects on how power is dissipated. In clocked systems, all the circuits are synchronized to switch simultaneously, thereby causing current surges on the power lines and problems with electro-migration. On the other hand, asynchronous circuits have evenly distributed power demands.

The main argument against delay-insensitive and speed-independent circuits is low area efficiency. Synchronous circuits generally require less area. But as scaling provides us with more area, the limiting factor becomes design complexity rather than area efficiency. Delay-insensitive systems allows the use of a hierarchical design methodology free from the lower level complexities of wire and module delays. There presently exist 0.1 micron transistors [Sai87], and further technological advances are inevitable, regardless of what technology is used. The possibility of biochips [Cart87] would bring us more area than we can handle. Closer to present technologies are quantum transistors [Bate88], where the significant differences in wire and gate delays would tend towards using finer grain delay-insensitivity.

There has also been thoughts of using optical and superconducting interconnects to eliminate the wire delay problem. High temperature

superconductors appear to be advancing rapidly. But even these have their limits. To travel 1 meter at the speed of light, it takes 3.3ns (1 / $3x10^8$m/s). Possible gate speeds are already well below 100ps, such as room temperature CMOS/SOI gates reported to operate at as low as 30ps [Vasu88]. Superconducting gates can operate at below 10ps. Even at the speed of light, interconnect delays will have to be considered.

# References


**[Bate88]** R.T. Bate, "The Quantum-Effect Device: Tomorrow's Transistor?" *Scientific American*, Vol. 258, No. 3, Mar. 1988, pp. 96-100.

**[Bert88]** C. Berthet and E. Cerny, "An Algebraic Model for Asynchronous Circuits Verification," *IEEE Trans. on Computers*, Vol. 37, No. 7, July 1988, pp. 835-847.

**[Blac86]** D.L. Black, "On the Existence of Delay-Insensitive Fair Arbiters: Trace Theory and Its Limitations," *Journal of Distributed Computing*, Vol. 1, 1986, pp. 205-225.

**[Burn88]** S.M. Burns and A.J. Martin, "Syntax-directed Translation of Concurrent Programs into Self-timed Circuits," *Proc. 1988 MIT Conf. on VLSI*, 1988, pp. 35-50.

**[Cart87]** F.L. Carter ed., *Molecular Electronic Devices II*, Marcel Dekker, Inc., New York, 1987.

**[Chan73]** T.J. Chaney and C.E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE Trans. on Computers*, Vol. C-22, No. 4, April 1973, pp. 421-422.

**[Chap84]** D.M. Chapiro, *Globally-Asynchronous Locally-Synchronous Systems*, PhD thesis, Stanford University, Oct. 1984.

**[Chu87]** T.-A. Chu, "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications," *Proc. Int'l Conf. Computer Design*, 1987, pp. 220-223.

**[Chu87b]** T.-A. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*, Ph.D. thesis, Dept. of EECS, MIT, May 1987.

**[Dall87]** W.J. Dally and P. Song, "Design of a Self-Timed VLSI Multicomputer Communication Controller," *Proc. Int'l Conf. Computer Design*, 1987, pp. 230-234.

**[Dill88]** D.L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits," *Proc. 1988 MIT Conf. on VLSI*, 1988, pp. 51-65.

**[Dill88b]** D.L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits," Ph.D. thesis, Carnegie Mellon, Feb. 1988.

**[Hoar78]** C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, Vol. 21, No. 8 Aug. 1978, pp. 666-677.

**[Hoar85]** C.A.R. *Hoare, Communicating Sequential Processes*, Prentice-Hall, 1985.

**[Kell74]** R.M. Keller, "Towards a Theory of Universal Speed-Independent Modules," *IEEE Trans. on Computers*, Vol. C-23, No. 1, Jan. 1974, pp. 21-33.

**[Klee87]** L. Kleeman and A. Cantoni, "Metastable Behavior in Digital Systems," *IEEE Design and Test of Computers*, Dec. 1987, pp. 5-19.

**[Lam88]** P. Lam and H.F. Li, *A Design Methodology for the Hierarchical Composition of Fine-Grained Delay-Insensitive Systems*, Technical Report CCSD-VLSI-88-1, Dept. of Computer Science, Concordia University, April 1988.

**[Lam88b]** P. Lam and H.F. Li, "Hierarchical Design of Delay-Insensitive Systems," *Canadian Conf. on VLSI*, Oct. 1988, pp. 414-423.

**[Lam89]** P.N. Lam and H.F. Li, "Hierarchical Design of Delay-Insensitive Systems," to appear in *IEE Proceedings-E, Computers and Digital Techniques.*

**[Mari81]** L.R. Marino, "General Theory of Metastable Operation," *IEEE Trans. on Computers*, Vol. C-30 No. 2, Feb. 1981, pp. 107-115.

**[Mart85]** A.J. Martin, "The Design of a Self-timed Circuit for Distributed Mutual Exclusion," *Proc. 1985 Chapel Hill Conf. on VLSI*, H. Fuchs ed., Computer Science Press, 1985, pp. 245-260.

**[Mart86]** A.J. Martin, "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits," *Journal of Distributed Computing*, Vol. 1, 1986, pp. 226-234.

**[Mart86b]** A.J. Martin, "Self-timed FIFO: an Exercise in Compiling Programs into VLSI Circuits," *From HDL description to guaranteed correct circuit design*, D. Borrione ed., North-Holland, 1986.

**[Mart87]** A.J. Martin, "A Synthesis Method for Self-timed VLSI Circuits," *Proc. Int'l Conf. Computer Design*, 1987, pp. 224-229.

**[Mill65]** R.E. Miller, "Speed Independent Switching Theory," *Switching Theory*, Vol. II, Chapter 10, John Wiley & Sons, New York, 1965.

**[Moln85]** C.E. Molnar, T.-P. Fan, and F. U. Rosenberger, "Synthesis of Delay-Insensitive Modules," *Proc. 1985 Chapel Hill Conf. on VLSI*, H. Fuchs ed., Computer Science Press, 1985, pp. 67-86.

**[Nies88]** C. Niessen, C.H. van Berkel, M. Rem, and R. Saeijs, "VLSI Programming and Silicon Compilation; a Novel Approach from Philips Research," *Proc. Int'l Conf. Computer Design: VLSI in Computers & Processors*, 1988, pp. 150-151.

**[Owic82]** S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs", *ACM Trans. on Programming Languages and Systems*, Vol. 4, No. 3, July 1982, pp. 455-495.

**[Pete81]** J.L Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, 1981.

**[Prat82]** V.R. Pratt, "On the Composition of Processes," *Proc. Ninth Annual ACM Symp. Principles of Programming Languages*, Jan. 1982, pp. 213-223.

**[Prat86]** V.R. Pratt, "Modelling Concurrency with Partial Orders," *International Journal of Parallel Programming*, Vol. 15, No. 1, 1986.

**[Prob88]** D.K. Probst and H.F. Li, *Abstract Specification, Composition, and Proof of Correctness of Delay-Insensitive Circuits and Systems*, Technical Report CCSD-VLSI-88-2, Dept. of Computer Science, Concordia University, April 1988.

**[Reis85]** W. Reisig, *Petri Nets: An Introduction*, Springer-Verlag, 1985.

**[Rem83]** M. Rem, J. van de Snepscheut, and J.T. Udding, "Trace Theory and the Definition of Hierarchical Components," *Third Caltech Conf. on VLSI*, 1983, pp. 225-239.

**[Rose88]** F.U. Rosenberger, C.E. Molnar, T.J. Chaney, T.-P. Fang, "Q-Modules: Internally Clocked Delay-Insensitive Modules," *IEEE Trans. on Computers*, Vol. 37, No. 9, 1988, pp. 1005-1018.

**[Sai87]** G.A. Sai-Halasz, M.R. Wordeman, D.P. Kern, et. al., "Experimental Technology and Characterization of Self-Aligned 0.1um-Gate-Length Low-Temperature Operation NMOS Devices," *IEEE Electron Devices Meeting*, 1987, pp. 397-400.

**[Sara82]** K.C. Saraswat and F. Mohammadi, "Effect of Scaling of Interconnections on the Time Delay of VLSI Circuits," *IEEE Journal of Solid-State Circuits*, Vol. SC-17, No 2, April 1982, pp. 275-280.

**[Seit80]** C.L. Seitz, "System Timing," in Mead & Conway, *Introduction to VLSI Systems*, Chapter 7, Addison-Wesley, 1980.

**[Snep85]** J. van de Snepscheut *Trace Theory and VLSI Design*, Springer-Verlag Lecture Notes on Computer Science, vol. 200, 1985.

**[Uddi84]** J.T. Udding, *Classification and Composition of Delay-Insensitive Circuits*, Ph.D. thesis, Eindhoven University of Technology, 1984.

**[Uddi86]** J.T. Udding, "A Formal Model for Defining and Classifying Delay-Insensitive Circuits and Systems," *Journal of Distributed Computing*, Vol. 1, 1986, pp. 197-204.

**[Unge69]** S. Unger, *Asynchronous Sequential Switching Circuits*, John Wiley & Sons, 1969.

**[Vasu88]** P.K. Vasudev, K.W. Terrill, and S. Seymour, "A High Performance Submicrometer CMOS/SOI Technology Using Ultrathin Silicon Films on SIMOX," *Symposium on VLSI Technology*, 1988, pp. 61-62.

**[Wagn88]** K.D. Wagner, "Clock System Design," *IEEE Design and Test of Computers*, Oct. 1988, pp. 9-27.