National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Service | Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

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.

NL 339 (r 88/04) c

Canada

High-Level Synthesis of

Bundled Delay-Insensitive Circuits from

Occam Program Specifications using Time-sharing

K.J.Venkatesh

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal , Quebec , Canada

February 1992

© K.J. Venkatesh , 1992

National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Service | Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-73692-5

Canada

Abstract

High-level Synthesis of
Bundled Delay-Insensitive Circuits from
Occam Program Specifications using Time-sharing

K.J.Venkatesh

This thesis presents techniques for synthesizing bundled delay insensitive circuits from high-level occam programs using time-sharing to reduce the area of synthesized circuits. A semantic model, called intermediate form (IFORM), is presented for a subset of occam. Syntax directed translation from a occam program specification into an IFORM, and from IFORM into bundled Delay-Insensitive circuits are presented. The idea of time-sharing is used in the translation. Time-sharing refers to the use of the same circuit to carry out the computation described by different instances of an operation in a program. To improve the degree of time-sharing, the control flow of the given program is changed by performing augmentations on the intermediate form of a given program. Augmentation by data flow extraction to increase concurrency has been presented. The intermediate form is also used for the performance evaluation of the translated circuit. Some discussion on the applicability and limitation of previously published performance evaluation methods for asynchronous systems is provided. Finally, a strategy for exploring the optimization search space is outlined. The technique is applied to the synthesis of an elliptic filter. A software program has been developed to implement the synthesis methodology presented in this thesis.

ACKNOWLEDGEMENTS

I would like to thank Dr. R. Jayakumar, my thesis supervisor and Prof. H.F. Li for all their help, understanding, and guidance. Their constructive criticisms and keen ability to pinpoint important issues and weaknesses have been a major part in the shaping of this thesis. I would also like to thank the members of the DI group S.C.Leung and Ping N. Lam.

Table of Contents

List of Figures

Figure 1:	An implementation of $(f(a) \wedge f(b))$ using time-sharing	**7**
Figure 2.1:	Input operation	**14**
Figure 2.2:	Output operation	**15**
Figure 2.3:	Assignment	**16**
Figure 2.4:	SEQ construct	**17**
Figure 2.5:	PAR construct	**17**
Figure 2.6:	WHILE construct	**18**
Figure 2.7:	IF construct	**18**
Figure 2.8:	ALT construct	**19**
Figure 2.9:	Circuit for realizing the communications over channel a between multiple input/output processes	**21**
Figure 3.1:	An example occam program and its parse tree	**29**
Figure 3.2:	Circuit for realizing Time-Sharing	**30**
Figure 3.3:	Circuit for realizing Time-Sharing	**31**
Figure 4.1:	Augmentation by data flow extraction	**33**
Figure 4.2:	Augmentation by serialization	**34**
Figure 4.3:	Illegal augmentation	**35**

Figure 4.4: Circuit for realizing Time-sharing 36

Figure 4.5: Merging of Input/Output transitions 38

Figure 4.6: Unfolding of loops for merging Input/Output transitions 38

Figure 5.1: Occam specification of an Elliptic filter 51

Figure 5.2: IFORM for the Elliptic filter 54

Figure 5.3: The layout of the optimized elliptic filter 58

Figure 5.4: Augmenting the IFORM by serialization 59

Figure 5.5: EDIF Netlist of the circuit with four multiplier modules 60

Figure B.1: Fork 78

Figure B.2: C-element 78

Figure B.3: XOR 79

Figure B.4: Toggle 79

Figure B.5: XOR 79

Figure B.6: Select 80

Figure B.7: Call 80

Figure B.8: Bcall 81

Figure B.9: Arbiter 82

Figure B.10: ATS 83

Figure B.11: De-multiplexer 83

List of Tables

Table 1: Synthesis results **50**

Table 2: HAL synthesis results **51**

1. Introduction **1**

1.1 High-level synthesis 1

1.2 Delay-Insensitive (DI) circuits 3

1.3 Bundled Delay-Insensitive circuits 4

1.4 Specification 4

1.5 Intermediate form 5

1.6 Translation 5

1.7 Performance evaluation of Asynchronous concurrent systems 8

1.8 Analysis of Distributed Communicating Processes 9

1.9 Contribution of the Thesis 9

2. Intermediate Form **11**

2.1 Requirements 11

2.2 The Intermediate Form 13

2.3 Relating Program and Intermediate Form 14

2.4 Relating Intermediate Form and Circuit 21

2.5 Correctness of Implementation 24

3. Time-Sharing **26**

3.1 Abstract Requirements 26

3.2 Time-sharing Conditions in Intermediate Form 26

3.3 Procedure for Identifying Time-Sharing Candidates 28

3.4 Circuit for Realizing Time-Sharing 30

4. Augmentation, Performance Evaluation, and Optimization Strategy **32**

4.1 Augmentation by data flow extraction 32

4.2 Augmentation by serialization 33

4.3 Performance Evaluation 37

4.4 Optimization Strategy 38

5. Software implementation **41**

5.1 The Elliptic Filter Example 50

6. Conclusion **65**

References **67**

Appendix A: Occam Subset: Syntax and Sematics **71**

Appendix B: Basic circuit building blocks **78**

Appendix C: Basic DI Cell Library **83**

Chapter 1

Introduction

As VLSI systems become larger and more complex, managing their complexity becomes a major part of the design process. One method of taming their complexity is to use automatic (systematic) methods for generating circuits from behavioral descriptions. This allows the designer to abstract away details of the low-level circuits and think of system behavior in terms of high level programs. In recent years there has been a trend toward automating at higher levels of the design hierarchy. There are a number of reasons for this: (1) shorter design cycle: if more of the design process is automated, a design can be completed faster, which in turn can lower the cost significantly; (2) fewer errors: if synthesis process can be verified to be correct - by no means a trivial task - there is a greater assurance that the final design will correspond to the initial specification; (3) the ability to search the design space: a good synthesis system can produce several designs from the same specification in a reasonable amount of time, which allows the developer to explore different trade-offs between cost, speed, power etc. in choosing one of several designs produced by the automated design system. In this thesis, a systematic procedure to synthesize VLSI circuits from the behavioral description of a system is presented.

1.1. High-level Synthesis

The synthesis task is to take a specification of the required behavior of a system and a set of constraints and goals to be satisfied, and to find a circuit structure that implements the behavior while satisfying the goals and constraints. By *behavior* we mean the way the system or its components interact with the environment, i.e. the mapping from inputs to outputs of the system. *Structure* refers to the set of interconnected components that make up the system, typically described by a netlist. Ultimately, the structure must be mapped into a *physical design*, that is, a specification of how the system is actually to be built. Behavior, structure, and physical design are usually distinguished as the three

domains in which hardware can be described.

Just as designs can be described at various levels of detail, so can synthesis take place at various levels of abstraction. The hierarchy of levels [1] generally recognized as applicable to digital designs are system, algorithm, register-transfer, logic and circuit. High-level synthesis, as we use the term, means going from an algorithmic level specification to a register-transfer level structure. The specification, at the algorithmic level, gives the required mapping from sequences of inputs to sequences of outputs, where those inputs and outputs may communicate with the outside environment or with another system-level component. The specification should constrain the internal structure of the system to be designed as little as possible. From the specification, the synthesis system produces a description of a data path., that is, a network of registers, functional units, multiplexers, and buses. If the control is not integrated into the data path, the synthesis system must also produce the specification of the control part.

Usually there are many different structures that can be used to realize a given behavior. One of the tasks in synthesis is to find the structure that best meets the constraints, such as limitations on time, area, or power, etc. while minimizing other costs. For example, the goal might be to minimize area while achieving a certain required processing rate.

While most of the work in high-level synthesis has been done for synchronous systems, there is a considerable interest to move away from totally synchronous systems to self-timed asynchronous systems. As VLSI systems become larger, clock skew becomes a more serious problem. Using a longer clock period alleviates the clock skew problem; but the overall system throughput will be reduced. *Self-timed signalling* [34] was advocated as an approach to VLSI systems design. However, asynchronous circuits are known to be difficult to design correctly [39]. One method for making asynchronous/self-timed systems easier to design is to use automatic (systematic) methods for generating circuits from behavioral descriptions.

1.2. Delay-Insensitive (DI) circuits

There are various forms of asynchronous circuits. *Delay-insensitive (DI)* circuits [29, 38] are the most robust form whose functional correctness is unaffected by delays in the components and by delays in wires connecting the components. Delay insensitive system are composed out of hierarchy of DI modules. The communication between two DI modules is based on self-timed signalling protocol, such as 2-phase or 4-phase handshaking. The 2-phase handshaking (also called 2-phase transition signalling) is based on transition signalling where either rising or falling signal transition signifies an event. The "two-phase ' part of this name indicates that only two phases of operation are distinguished: the sender's active phase and the receiver's active phase. An event terminates each phase: the request event terminates the sender's active phase, and the acknowledge event terminates the receiver's active phase. In 4-phase handshaking protocol, the first two phases are the same as the 2-phase handshaking protocol. The other two phases refers to the request and acknowledge signals returning to a low level before the next communication can take place between the sender and the receiver. *Speed-independent (SI)* circuits [12, 23, 24] are closely related to DI circuits in that they allow arbitrary component delays, but zero wire delays are assumed. Classical asynchronous circuits [39] require various delay compensations for correct operation. Because of the lack of design tools, they are usually designed to operate in the fundamental mode: only one input signal transition is allowed at a time and an input signal transition can be sent in only when the entire circuit has stabilized (no signal transition takes place within the circuit).

Other than purely asynchronous circuits, some approaches use local clocks within circuit modules while communications among circuit modules are asynchronous [33]. Systems using this approach are usually called *globally asynchronous locally synchronous systems*. *Selective clocking* approach [11, 26, 42] is another one that also involves the use of clocks. Clocking is done at *some* of the state transitions of a finite state machine specification of the circuit. Synchronous systems always have clocking when there is a state transition, while there are no clocking at all in state transitions for purely asynchronous circuits.

1.3. Bundled Delay-Insensitive circuits

Since truly DI implementations of data manipulating circuit modules, e., adder, multiplier, etc. require very large areas, a compromise between robustness and ar a-efficiency is adopted and such circuits are named as *bundled delay-insensitive circuits*. Data manipulating circuit modules are designed using classical boolean techniques. Data path from one circuit component to another is bundled with a control wire. When a circuit component produces a data value to be consumed by a receiving component, the data values is put onto the data path and a signal transition (rising or falling signal transition because in two phase handshaking, rising and falling signal transitions mean the same) is sent along the control wire. It is assumed that by the time the signal transition along the control line arrives at the receiving component, the data value at the data path is valid and the receiving component can consume that data value. This assumption can be guaranteed by introducing delay elements in the control line. The delay introduced by the delay element has to be more than the total delay of the gates in the data path. The producer of the data value must not produce another data value again until it receives an acknowledgment directly or indirectly (i.e. from some other circuit component other than the receiving component). The control circuit is a DI circuit.

1.4. Specification

The system to be designed is usually represented at the algorithmic level by a programming language such as Pascal[37] or Ada [14], or by a hardware description language that is similar to a programming language, such as ISPS[7], DSL[8], MIMOLA[43] or behavioral VHDL[9], CSP[5, 23], Occam[2, 3, 4, 41]. A realistic specification language should contain mechanisms to specify hierarchy, usually procedures, and a way of specifying concurrent tasks.

Essentially, asynchronous circuits are synthesized from two kinds of specifications, namely, signal transition level specifications and concurrent programming language (e.g. occam, CSP) specifications. Signal transition level specifications are more suitable for specifying control circuits inside systems. Data manipulating circuit components, e.g.

adder, have very big signal transition level specifications and are not feasible to be implemented using purely DI/SI circuits.

When a whole system is to be implemented, a high level concurrent programming language specification is more appropriate than a signal transition level specification. CSP/occam based concurrent languages are popularly used for such system description. Synchronous communication is required to achieve delay-insensitive communication between processes, and the communication primitive constructs provided by those languages captur exactly this requirement. The designers are free from worrying how to realize synchronous communication in terms of signal transitions.

In this thesis, the specification language is a subset of occam [17], a language based on CSP. An occam program is the description of a computation in terms of the order in which instances of some operations are performed. The operations include the primitive operations supported by the language, e.g. +, *, input/output operations, and user-defined functions/procedures.

1.5. Intermediate form

Since program and a circuit are at two different levels of abstraction there is a need to use an intermediate representation which forms a bridge between these two levels. The first step in high-level synthesis is usually the compilation of the formal language specification into an internal representation. Most approaches in synchronous design use graph-based representations that contain both the data flow and the control flow implied by the specification, although parse trees are also used [25]. In this thesis, an intermediate form is proposed (chapter 2) as a semantic model for occam programs and as a tool to examine the issues in time-sharing circuits among instances of the same operation in a program.

1.6. Translation

The simplest of all the translation methods is syntax-directed translation. Each of the syntactic construct is mapped into a corresponding circuit entity. The circuit entities are connected in the same way the syntactic constructs are composed in the program. Thus

in syntax-directed translation, the computation described by each of the instances of operations in a program is done by a physical circuit component. The circuit components in the translated circuit are activated in exactly the same order as the instances of the operations during the execution of the program.

Martin [23] at CalTech has succeeded in compiling specifications in a language based on CSP into gate-level circuits. Required program specifications are first decomposed into many smaller processes initiated by signals on new communication channels. These simpler processes are expanded to include details of the four phase handshake used for control signals and then mapped into a set of production rules that define when the handshake signals are set and reset. These production rules are strengthened to enforce sequencing, and then mapped into a library of gates to construct the circuit. Various ways of *shuffling* signal transitions to obtain different circuits were presented in [23]. In [5] a syntax-directed translation for compiling a CSP-like language specification into SI circuit is described. Translation of occam programs into *bundled DI circuits* [36] has been considered in [1, 3, 4, 41].

Syntax-directed translations of programs written in CSP like languages are presented in [1, 3, 4]. In [4] some post-translation optimizations by replacing sub-circuits with simpler sub-circuits is proposed. The optimizations are limited. Optimizations for reducing the areas of the synthesized circuits are also mentioned in [1, 3]. However, [1, 3, 4] did not give any systematic approach to carry out the optimizations, nor did they provide any evaluation on the space-time trade-off when such optimizations are carried out. A preliminary report on the work presented in this thesis is discussed in [41]. Sutherland demonstrated the synthesis of a special class of bundled DI circuits called *micropipelines* [36].

In [35] the synthesis of DI circuits from "synchronized transitions" which is similar to the UNITY [10] programming language is considered. However, correct operation of the synthesized circuits depend on the *isochronic fork* (the delays in the forked wires are the same) assumption [23].

Basic components for bottom-up design of DI circuits/systems is presented in [19,

33]. In [33] a basic module called the *Q-module* for designing globally asynchronous locally synchronous systems is presented. Each Q-module has a local clock while communication between Q-modules is asynchronous. Lam and Li [19] described a set of basic building blocks and an approach based on signal transition graphs for designing DI circuits.

In this thesis, the idea of *time-sharing* is used to reduce the area of the synthesized circuits. Time-sharing refers to the use of one physical circuit component to perform the computation described by different instances of an operation (the circuit component is said to be *time-shared* among the instances). Such time-sharing is considered because a major cause for the large area of circuits obtained by syntax-directed translation methods is that each instance of an operation in the program is mapped into a separate circuit component. For a big program, the area of the translated circuit will be prohibitively expensive. Fig. 1 is an example on time-sharing. The C-element makes sure that a transition will be initiated its output only after signal transitions occur at both "a" and "b" inputs. For the correct operation of the circuit, signal transitions at a and b should never happen together. The

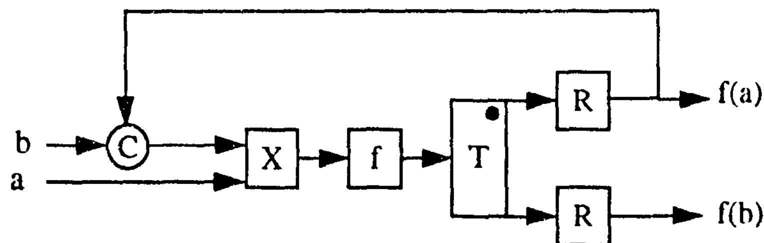


Fig. 1 An implementation of (f(a) ∧ f(b))
using time-sharing

evaluations of f(a) and f(b) are done by the same circuit module f. The multiplexer X multiplexes the bundled "a" and "b". The Toggle element initiates a signal transition, at one of its outputs alternatively, each time it receives a signal transition at its input. The two Register elements (R) latches the data values computed by the "f" circuit module for the data values "a" and "b". If the boolean function f() takes 1000 transistors to implement, then the saving will be about 20%. The circuit elements used for the control circuits are similar

to those in [36].

Given an occam program, an intermediate form representation of it is first obtained. Removal of some of the control sequencing in the original program is considered to improve concurrency in the resulting circuit. Instances of the same operation are then chosen to time-share a physical circuit component. Some concurrent operations may be serialized so that more time-sharing can be carried out provided that such serializations do not result in unacceptable performance degradation. An area optimization strategy based on a greedy algorithm is outlined. Finally, the optimization method is illustrated with a fifth order elliptic filter example.

Time-sharing will result in less hardware in the synthesized circuit if the overhead (additional hardware) for time-sharing is less than the hardware required to implement the operation. The removal of unnecessary control sequencing in the original program and the serialization of concurrent operations allow more time-sharing while minimizing the impact on reduction in throughput of the synthesized circuits.

1.7. Performance Evaluation of Asynchronous Concurrent Systems

Performance evaluation of asynchronous concurrent systems appeared in [6, 22, 30]. Ramamoorthy and Ho gave a performance measure called *cycle time* for decision free Petri nets [30]. The cycle time is the asymptotic average time between successive firings of a transition in the net. For a decision free Petri net, the cycle times for all the transitions in the net are the same. Given the firing time of every transition and the initial marking, a lower bound on the value of the cycle time can be estimated. Magott provided a tighter lower bound for the cycle time [22].

Burns and Martin [6] used an *event-rule system* for the performance evaluation of asynchronous circuits. A network of circuit components and the specification implemented by it are assumed given. To achieve the performance requirements of the overall circuit, the performance requirements on individual circuit modules are identified. These requirements are then used to guide the internal design of the circuit modules (e.g. sizes of transistors inside the circuit modules). Only a method for deterministic computation was

presented.

These performance evaluation methods are of limited use in the analysis of occam programs (and in general, network of processes that communicate synchronously). All of these methods assume the classical transition firing rule, i.e. when (and only when) all the input places of a transition are marked, the transition is fired and tokens are put at the output places. The dynamic behaviors arising from the constraint imposed by synchronous communication cannot easily be captured by the static structure of an analysis model for a given occam program, especially in the presence of non-determinism. Thus, some modifications of the published methods are required, or some new performance evaluation methods have to be developed. More on this aspect will be discussed in chapter 4.

1.8. Analysis of Distributed Communicating Processes

Approaches for data flow analysis of distributed communicating processes have been presented in [13, 27, 31]. The methods in [13, 27] are based on some invariant proof systems. Reif and Smolka [31] suggested *event spanning graphs* for analysis. However, constructs like the alternation statement in occam are not modeled. In this thesis an intermediate form is presented which models alternation statement and synchronous communication in occam. Data flow extraction is done on the intermediate form in such a way that the producer-consumer data dependency and the control sequencing among input and output transitions are preserved

1.9. Contribution of the Thesis

This thesis provides a semantic model, called intermediate form (IFORM), for a subset of occam programs (programs that do not contain the WAIT or STOP process). No such intermediate form was used in [1, 3, 4] to allow more systematic optimization of the synthesized circuit. Syntax-directed translations from a given occam program into an IFORM, and from the IFORM into bundled DI circuits are presented in chapter 2. While describing the intermediate form we also show how some of the stated requirements are met. Other requirements are shown to be met in chapters 3 and 4. By means of the

IFORM, a formal definition is given for correctness of implementation ("a circuit implements an occam program"). Some time-sharing results are presented and proven in chapter 3.

To improve the degree of time-sharing, the control flow of the given program is then changed by performing augmentations on the intermediate form of a given program. Some augmentation strategies are explored in chapter 4. The intermediate form is also used for the evaluation of the performance of the translated circuit. Some discussion on the applicability and limitation of previously published performance evaluation methods for asynchronous systems is provided. Finally, a strategy for exploring the optimization search space is outlined. A fifth order elliptic filter [14] is used in chapter 5 as a demonstration of the proven as well as the conjectured results.

Chapter 2

Intermediate form

An intermediate form is proposed to be the semantic model for a subset of occam programs. All the constructs in [17] are allowed except the WAIT and STOP processes. It is not possible to implement the WAIT process in DI circuits since the process behavior depends on the time-out mechanism. A STOP process will suspend the process for ever. Since we are interested in deriving hardware circuits which are non-terminating, STOP processes are excluded in the occam subset. To establish delay-insensitive communication between processes, the communication is required to be synchronous. Occam (or CSP-based languages) provide such synchronous communication primitives. Furthermore, occam has the alternation construct which allows interesting non-deterministic behaviors to be specified easily. As a result, occam-based or CSP-based languages are commonly used for describing concurrent systems that are to be implemented as DI circuits [1, 3, 4, 5, 41].

2.1. Requirements

The intermediate form is supposed to be a semantic model for occam as well as a tool for exploiting the space-time trade-off optimizations and proving correctness. Hence, it should have the following characteristics: (1) modeling of arbitration, (2) modeling of synchronous communication, (3) modeling of concurrency, (4) allowing the detection of transitions that can time-share a circuit, (5) allowing the detection of transitions between which augmentation can be performed, (6) evaluating the effect of time sharing and/or change in control flow on the performance of the synthesized circuit, and (7) proving correctness.

The justification for the above criteria on the intermediate form are as follows. As a semantic model for occam programs, the intermediate form must be able to capture the main features of occam programs, namely arbitration (alternation statement) and synchronous communication.

Two instances of an operation in a program can time-share a circuit if the two instances are performed sequentially under all possible behaviors of the program (note that the outcomes of choices can affect the way in which an occam program is executed). Some tool is needed to analyze the program behaviors (detection of operations that are executed sequentially under all possible program behaviors) in order to identify potential candidates to time-share circuits.

In time-sharing a circuit, additional multiplexing circuits are needed at the input and additional de-multiplexing circuits are needed at the output. For example, in fig. 1(page 7), there are data multiplexers (bundled with the XOR) at the inputs of the circuit that implements f(), and there is a toggle at the control output of the circuit. The execution time will also be increased. The trade-off between space and time has to be examined carefully. Also, the mapping from the instances of operations in a program into physical circuits has to be captured.

Another optimization aspect is whether to maintain the control flow specified in the original program or not. Removing some of the control flow in the original program allows faster execution of the program. There is a need to analyze the data dependency in the original program in order to find out which control sequencing can be removed. Serializing the execution of some concurrent operations allows them to time-share a circuit so that further savings in area can be achieved. The time-sharing example in fig. 1 involves the serialization of the two instances of f(). When such augmentations are carried out, the original execution semantics of the occam program must be preserved. Also, effects on the performance (according to some meaningful measures) due to augmentation have to be evaluated.

Finally, there are issues on correctness: what is meant by "a circuit implements an occam program"? Correctness can be established more easily for syntax-directed translation because the correctness of each of the translation rules (i.e. whether the execution in the circuit level faithfully preserves the execution semantics of the corresponding program construct) can be judged separately. When optimizations are involved, the mapping from programs to circuits is not one-to-one. Furthermore, occam programs and circuits are at

different levels of abstraction, and the behaviors of the circuit elements cannot be expressed in an occam program. As a result, there is a need to define correctness in terms of some intermediate form so that all optimization tricks can be proved correct.

2.2. The Intermediate Form

The proposed intermediate form is a marked graph [32] with modified semantics. Marked graph is a Petri net. A Petri net is a directed graph G = (P, T, E) with a set of vertices P, called places, a set of vertices T called transitions and a set of directed edges $E \subseteq (P \times T) \cup (T \times P)$ connecting places to transitions and vice-versa. The input and output sets for each place and transition are defined as

$$I(p_j) = \{t_i \mid (t_i, p_j) \in E\},$$
$$O(p_j) = \{t_i \mid (p_l, t_j) \in E\},$$
$$I(t_j) = \{p_l \mid (p_i, t_j) \in E\},$$
$$O(t_j) = \{p_i \mid (t_i, p_j) \in E\}.$$

A Petri net G = (P, T, E) is a marked graph if

$| I(pj) | = | O(pj) | = 1, \forall pj, \in P$. Thus in a marked graph each place has exactly one input transition and output transition. Marked graphs are able to explicitly represent concurrency expressed in the original program. Thus the intermediate form meets the requirement (3) stated in section 2.1. To account for the if-then-else statement, the while statement, and the alternation statement in an occam program, special kinds of transitions with specific firing rules are introduced to model the desired behaviors. To simplify the presentation, no vectors of variables or channels, or replicators (FOR) are used in the examples.

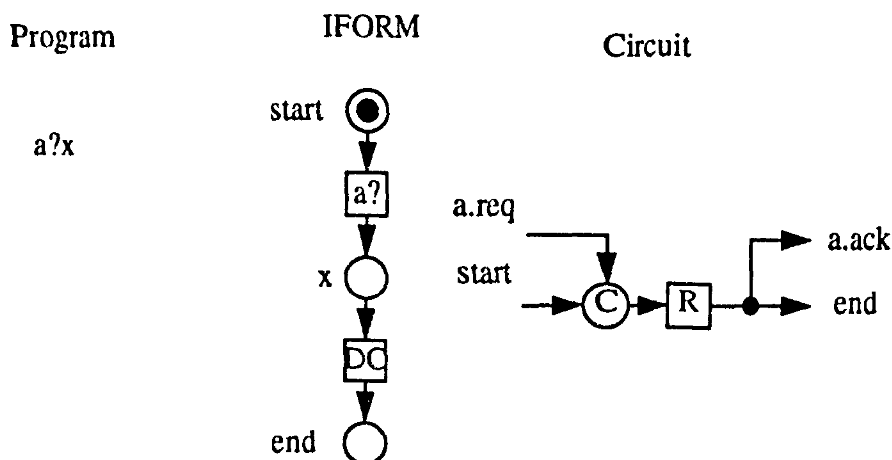
The syntactic constructs of a program are represented in the intermediate form by places, transitions and edges between places and transitions. Program behaviors are modeled by firings of transitions in the net. Similarly, the places, transitions, and edges correspond to physical circuits and interconnections among them. Firings of transitions correspond to activation of the circuit component represented by it. Occam has three types

of primitive processes, shown in fig. 2.1 - 2.3, which are composed to form a program using five types of control constructs, shown in fig. 2.4 - 2.8. Each of fig. 2.1 - 2.8 shows the primitive processes or how to compose processes using the control constructs, the intermediate form, and the corresponding circuit. Because each place immediately precedes only one transition, the places are not explicitly shown except for the initial (labeled "start") and final (labeled "end") places.

2.3. Relating Programs and Intermediate Forms

The intermediate forms for the primitive processes are considered first. An input process is represented as shown in fig. 2.1. The control point immediately preceding (following) the input process in the program is represented by a *control place* labeled "start" ("end") in the intermediate form. Execution of the input process involves an input operation over channel a and the return of a data value for variable x. The input operation is represented by a *data transition* labeled with the channel name (i.e., a?). That transition is called a data transition because firing of it produces a data value for some variable. The control bundled with that data value for x is represented by a *data place* labeled x.

Fig. 2.1 Input operation

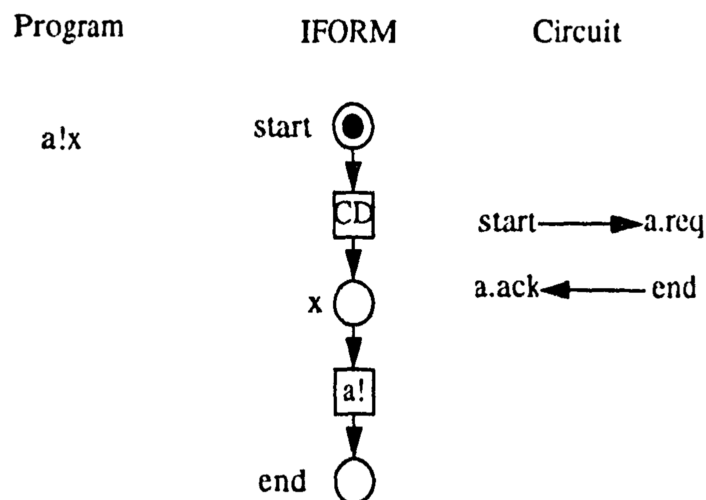


A token represents program control or data availability. When a control place is marked with a token, it means that the execution of the program has come to the control point represented by the control place. When a data place is marked with a token, it means that the data value represented by the control place is available. The control bundled with

the data value obtained as a result of the input operation is used to generate the control following the input process using a *data-to-control transition* (transition labeled "DC" in the intermediate form).

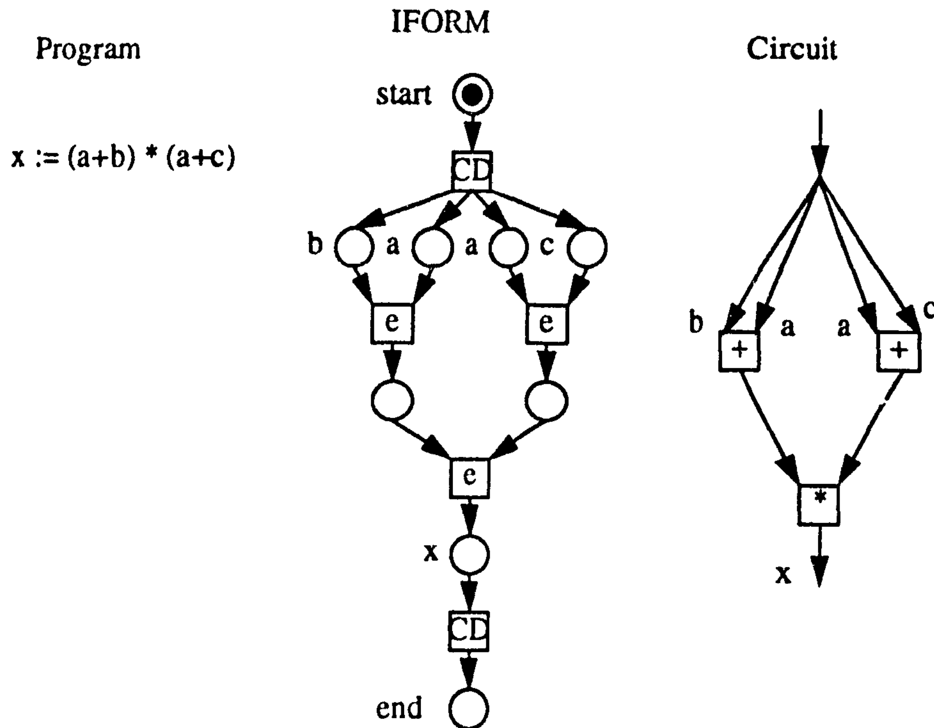
An output process is represented as shown in fig. 2.2. The control point immediately preceding (following) the output process in the program is represented by a control place labeled "start" ("end") in the intermediate form. Execution of the output process involves an output operation over channel a and the sending of a data value of variable x. The output operation is represented by a data transition labeled with the channel name (i.e., a!). That transition is a data transition because firing of it consumes a data value of some variable. The control bundled with that data value of x is represented by a data place labeled x. It is generated from the control preceding the output process using a *control-to-*

Fig. 2.2 Output operation



data transition (transition labeled "CD" in the intermediate form).

Fig. 2.3 Assignment



For an assignment statement, the expression on the right hand side is expanded into a "data flow graph". Fig. 2.3 is an example. Each transition, which is a data transition, represents a data manipulating operation in the expression. A data transition is labeled by the operation, e.g. +. Input and output places of the data transition are data places. A data place between two data transitions represents the value produced by one transition to be consumed by the other transition. If a data place represents some value of a variable, then it will be labeled by the name of the variable. When the data place is marked with a token, it means the data value is produced and is ready to be consumed. A control-to-data transition is used to generate from the control enabling the assignment statement the controls needed to bundle with the data values needed to activate the network of data transitions. Likewise, a data-to-control transition is used to generate from the controls bundled with the data values eventually produced by the network of data transitions the control representing the completion of the execution of the assignment statement.

Data places are introduced as a means to exploit the data dependency among data

transitions obtained from expanding primitive processes, especially when they are composed by a sequence construct. Extracting the data dependency relation [18] between these data transitions allows more concurrency. The anti-data-dependency and output dependency relations [18] can be ignored because values of the same variable are produced by distinct circuit components (if no time-sharing is applied) and are not written into a register for storing the values of that variable. When time-sharing is applied, production of values corresponding to firings of different data transitions in the intermediate form will be carried out by the same physical circuit. Appropriate control circuits will be used to route, from the output of the physical circuit, the values that correspond to the firings of the same data transition (different data transitions) to the same register (different registers) for storage. Consequently, only the data dependency relation between the data transitions needs to be considered.

Composition of processes using the five types of control constructs is now considered. *Control transitions*, which represent the four basic control operations, Branch (B), Fork (F), Join (J), and Merge (M), are used to "glue" the intermediate forms for the processes being composed. The processes are composed from primitive processes or composite processes using the types of control constructs. A transition labeled Q in fig. 2.6 - 2.8

Fig. 2.4 SEQ construct

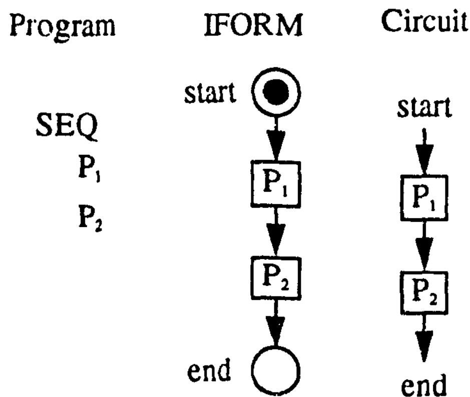
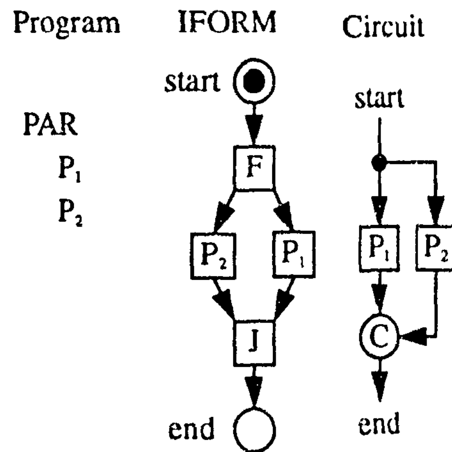


Fig. 2.5 PAR construct



represents the network of data transitions corresponding to the boolean expression Q in

the program. Edges between control places and transitions represent the control flow in the program.

Fig. 2.6 WHILE construct

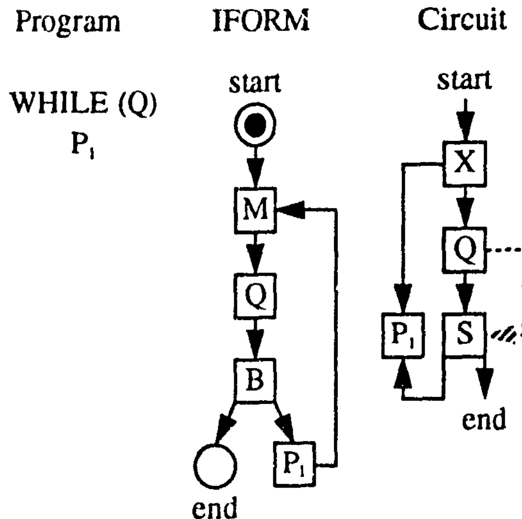
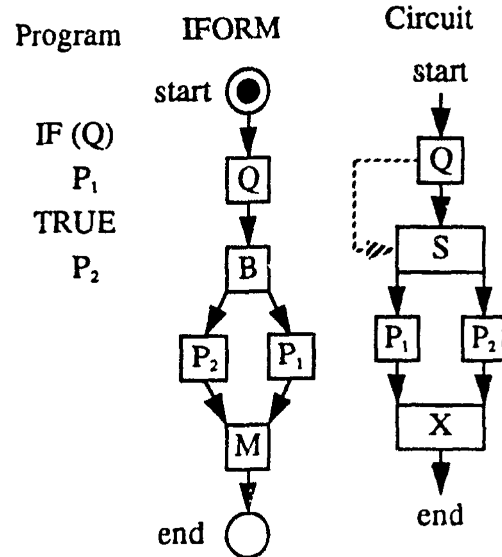


Fig. 2.7 IF construct



A distinct feature of occam is the ALT statement. Each ALT statement consists of several guarded commands (blocks of statements). A guarded command is enabled if the boolean expression in the guard is true and the input channel is ready. One and only one of the enabled commands will be executed. Each guarded block of statements is represented by transitions in the intermediate form. For each block, when all the statements in the block are expanded fully (i.e. even the primitive processes are also expanded and no further expansion can be done) the first transition is named the *entry (N) transition* and the last transition is named the *exit (X) transition*, as shown in fig. 2.8. Such explicit naming of entry and exit transitions avoids the use of a shared place and the firing rule to simulate the mutual exclusion among blocks. This results in a clearer flow relations among the entities in the intermediate form. The intermediate form thus meets the requirement (1) of IFORM (see section 2.1).

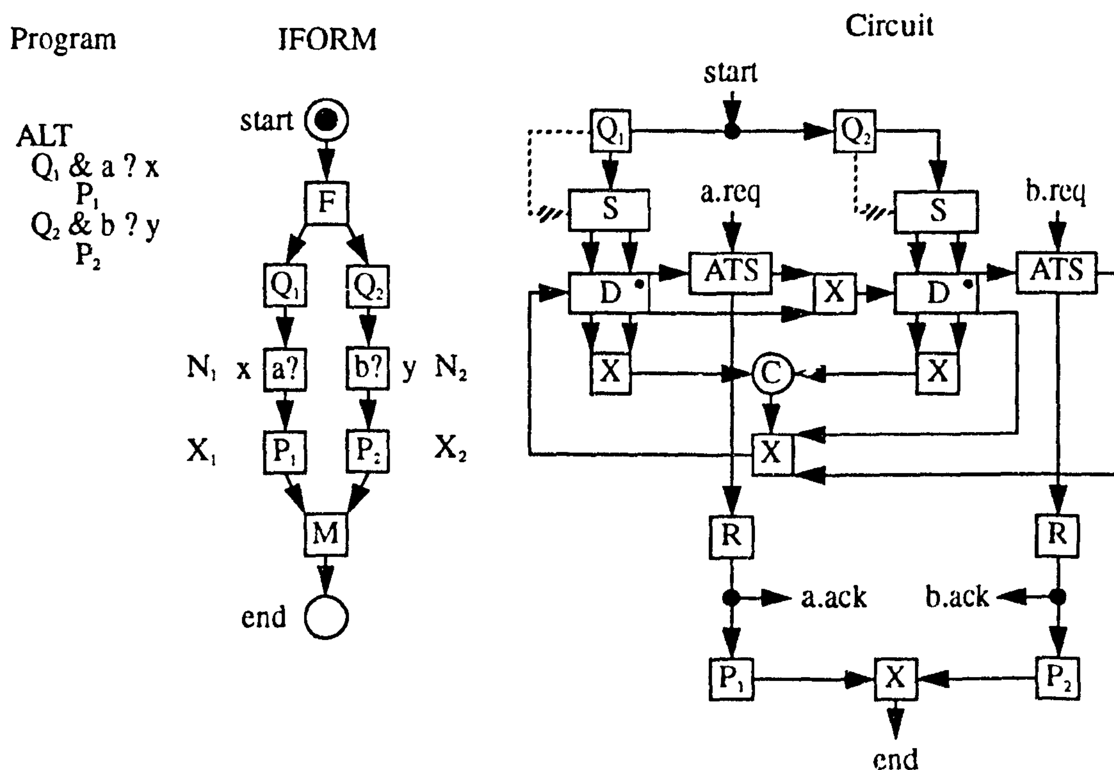
The intermediate form captures the static structure of a given program. The dynamic behaviors of a program are captured by transition firings in the intermediate form. Firing rules of transitions (except the entry transitions of ALT and the transitions that represent input/output processes) follow those in Petri nets. Only one token at one of

the input control places separated by ⊕ is needed to fire a transition. Only one of the output control places separated by ⊕ will have a token after the firing of a transition.

Consider the intermediate form of the WHILE construct (fig. 2.6). The merge control operation (the transition labeled by M) is fired when there is a token at the control place "start" or at the control place between the transitions labeled M and P_1. The transition labeled Q is then fired, which corresponds to the evaluation of the predicate Q. After that, either a token is deposited at the control place "end" or at the control place between the transition labeled B and the transition labeled P_1, depending on the boolean value of the evaluated predicate. Firing of the transition labeled P_1 results in the predicate Q being evaluated again.

For the entry transitions from the same ALT statement, when *all* the input control places are marked, one and only one of the entry transitions that have the corresponding predicates evaluated to be true will be fired and the tokens at the input places of *all* the entry transitions will be removed.

Fig. 2.8 ALT construct



To model the behavior of synchronous communication, the transitions representing the input and output processes communicating over the same channel must be fired together. Thus, even if a transition representing an input process (note that there can be more than one input process in the given occam program communicating over the same channel) communicating over a channel is enabled (i.e. the input control place of that transition is marked with a token), that transition cannot be fired until an output process communicating over the same channel (similarly, more than one output process in the given occam program can communicate over the same channel) is also enabled, and vice versa. When an input process and an output process that communicate over the same channel are enabled, the transitions representing them are fired together. If an input/output process communicates over a channel with the environment of the system described by the program, then, assuming the environment of the system is ready to communicate, firings of transitions representing those processes can be performed as soon as they are enabled. Thus the IFORM models synchronous communication which is one of the requirements of an intermediate form.

2.4. Relating Intermediate Forms and Circuits

The mapping into circuits from the intermediate forms of the primitive processes of occam is first considered. A data place represents a control line (a physical wire). The control line is bundled with the data path carrying the data value of the variable whose name is used to label the data place. A token corresponds to a signal transition. When a token is at a data place, it means a (rising or falling) signal transition takes place at the control line represented by the place. Edges between places and transitions represent the control lines (as well as the data paths) connected between sub-circuits. A data transition, e.g. labeled by +, corresponds to a circuit that performs that function, e.g. adder. A network of data transitions is mapped directly into a network of circuit components of corresponding functionalities. A control-to-data transition is mapped into a fork of appropriate number of outputs, and a data-to-control transition is mapped into a C-element of appropriate number of inputs.

Transitions representing input/output processes require special treatment. A chan-

nel is implemented using two wires. Synchronous communication between the two ends of a channel is accomplished by means of handshaking. One end (input or output) takes the initiation while the other end waits. The end that initiates (waits) is called *active* (*passive*). For each channel, either one of the two, (input passive, output active) or (input active, output passive), has to be determined. The circuits shown in fig. 2.1 - 2.2 are of the former choice. A passive input operation is carried out when there is an initiation from the corresponding output operation (a signal transition at the wire a.req) and the input operation is enabled (a signal transition at the wire start), which are acknowledged (a.ack and end) when the data value is received. In this thesis, it is assumed that all output processes are active (i.e. they initiate the communication) and all input processes are passive. Since transition signalling [36] is used, one transition at each of a.req and a.ack will complete a channel operation.

Consider two processes P_1 and P_2 that communicate over channel a. If there is only one input (output) process communicating over a in P_1 (P_2), then the two circuits in fig. 2.1 - 2.2 can just be put together to realize the synchronous communication between P_1 and P_2. If there are multiple such processes in P_1 and P_2 (the number of input processes in P_1 may be different from the number of output processes in P_2), then additional control circuit will be needed, as shown in fig. 2.9. Note that in fig. 2.9 P_1 consists of three output

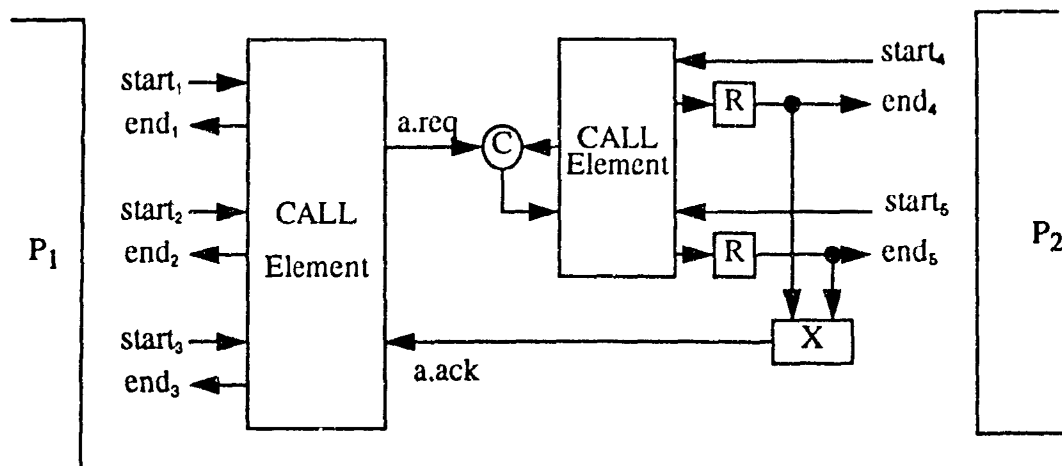


Fig. 2.9 Circuit for realizing the communications over channel a between multiple input/output processes

processes and P_2 consists of two input processes where $start_i$ and end_i (i=1, 2, 3) denote the starts and ends of the output processes in P_1, and $start_j$ and end_j (j=4, 5) denote the starts and ends of the input processes in P_2. The CALL element is from [36], and it allows the output processes that initiate (the input processes that wait for) the communication to be properly acknowledged.

The mapping of composition constructs into circuits is now considered. A control place represents a control line (a physical wire). When a token is at a control place, it means a (rising or falling) signal transition takes place at the control line represented by the place. Edges between places and transitions represent the control lines connected between sub-circuits. The control transitions labeled B (branch), F (fork), J (join), and M (merge) correspond to the circuit elements select, wire-fork, C-element, and XOR, respectively. Behaviors of these circuit elements can be found in appendix A and in [36].

The circuits shown in fig. 2.4 - 2.8 are the control circuits for the corresponding intermediate forms. Consider fig. 2.6. A merge transition (labeled M) is mapped into an XOR gate which produces an output signal transition when there is a signal transition at either one of the inputs. For safe use of the XOR gate, there cannot be two input signal transitions without an output transition in between. The transition Q is mapped into a circuit that evaluates the predicate. The circuit is also labeled Q. The branch transition (labeled B) is mapped into a select circuit element which generates a signal transition at one of its outputs from the control signal from Q, depending on the boolean value evaluated by the circuit obtained from Q. The use of the boolean value is shown by the dashed line. The control corresponding to a "true" boolean value from Q is used to activate the circuit block for P_1 while the other control signals the end of execution of the while loop. After the computation described by P_1 is finished, the predicate is evaluated by the circuit labeled Q.

The implementation of mutual exclusion among blocks of transitions is shown in fig. 2.8. The boolean expressions in the guards are evaluated by circuit blocks B_1 and B_2. A "token ring" similar to that in [3] is used to select the command whose boolean expression in the guard is true and the associated input channel is ready. The Arbitrating Test and

Set (ATS) block is the same as that in [3]. The de-multiplexer is the same as that in [21]. Behaviors of these circuit elements can be found in appendix A.

For data transitions obtained from processes composed by the five types of control constructs, data paths are connected according to the producer-consumer relations on the data values involved. A data value produced by a circuit component may be consumed by different circuit components, depending on the transition firings in the intermediate form. However, the data outputs of the producer circuit component are connected to the data inputs of *all* potential consumer circuit components. The dynamic behavior is captured in the control circuit which will generate the appropriate control signal to the consumer circuit component when the system is activated (i.e. when transitions in the intermediate form are fired or when the program is run).

On the other hand, a consumer circuit component may receive data value produced by different circuit components, depending on the transition firings in the intermediate form. Appropriate circuits, e.g. data multiplexers, are used to bridge between the data outputs of the potential producer circuit components and the data inputs of the consumer circuit component. The dynamic behaviors are also captured in the control circuit. When the system is activated, control signals will be generated appropriately to route the data value produced, through the data routing logic, e.g. data multiplexer, to the consumer circuit component.

2.5. Correctness of Implementation

Previously published synthesis methods [1,3,4,5] for occam or CSP-like languages did not address the correctness directly. Most of them are syntax-directed translation methods. The correctness of the synthesized circuit is built upon the correctness of the translation rules. The correctness of each translation rule is established by checking intuitively that the execution semantics of the program construct are carried out faithfully in the circuit level. As a result, not much optimization can be carried out as it is difficult to check that a significant change in the circuit level preserves the execution semantics to be realized.

To achieve more significant optimizations, there is a need to formally prove that the

optimized circuit can still execute whatever is described in the original program. The intermediate form proposed in this thesis serves such a purpose. Optimizations are carried out in the intermediate form level. Correctness can be established by showing that the intermediate form for the optimized circuit (which is obtained by translating the intermediate form using the rules stated in chapter 4) faithfully preserves the execution semantics of the intermediate form that represents the original program. Since the intermediate forms are in the same level of abstraction, it is possible to define a correctness criteria. The requirements on the correctness criteria are that if the conditions stated in the correctness criteria are valid, then one intermediate form (from which a circuit is obtained) will preserve the execution semantics of another intermediate form (which represents the given program). Chapter 4 gives the relation between programs and intermediate forms and the relation between intermediate forms and circuits. Correctness of the two relations can be judged intuitively. Thus, it is possible to assert that some circuit implements an occam program.

Moreover, since the intermediate form is in a higher level of abstraction than circuit level, more global optimizations can be examined than those presented in [4] which considered optimizations in the circuit level. On the other hand, at the intermediate form level, the freedom to change the control flow originally specified in the program and the ability to apply time-sharing allow more control over area/performance efficiency of the synthesized circuit than optimizations performed in the program level where the optimized program remains a valid occam program.

The correctness criteria in the intermediate form level are given below. A *run* of an intermediate form is a sequence of data transition firings in the intermediate form. A sequence of data transition firings can be obtained abstracting away all non-data transitions from a sequence of transition firings in the intermediate form. Firings of transitions forming a synchronous communication are replaced with a single communication transition in the sequence. Given two intermediate forms A and B, and there is a bijective mapping f between the sets of data/communication transitions in A and B, A is said to *realize* B if (1) every run of A is isomorphic to (under f) a run of B, and (2) for every run of B in which a data/communication transition is fired k (k can be finite or infinite) times, there

exists a run of A in which the corresponding data transition (under f) is fired exactly k times. A bundled DI circuit is said to *implement* an occam program if the intermediate form from which the circuit is obtained realizes the intermediate form of the program. For example, it will be shown later that the IFORM (b) in Fig. 4.2 is an implementation of the IFORM (a). Whereas IFORM (b) in Fig. 4.3 is an incorrect implementation of IFORM (a). Thus the IFORM meets the requirement on proving correctness.

Chapter 3

Time-Sharing

In this chapter, conditions for time-sharing a circuit between two data transitions in a given intermediate form are considered. Extension to multiple data transitions is straightforward.

3.1. Abstract Requirements

Consider the execution space. Two instances of an operation are said to occur *serially* if the values produced as a result of executing one instance of the operation are all consumed before every data value required for executing the other instance of the operation is produced. If the instances occur serially, then it is possible to time-share a physical circuit among them.

Consider two data transitions T_1 and T_2 (that represent the same type of operation P) in an intermediate form. When transition firings are allowed to occur from the initial marking of the intermediate form, each of T_1 and T_2 will be fired a number of times. Each firing of T_1 or T_2 corresponds to an instance of P in the execution space. If all the instances of P in the execution space due to firings of T_1 or T_2 are to time-share a circuit, then it will be required that execution of any two instances of the operation, regardless of whether the two instances are due to firing T_1 twice, T_1 once and T_2 once, or T_2 twice, occur serially. Transitions T_1 and T_2 are said to be able to *time-share* a circuit if all the instances of P due to firing of T_1 or T_2 occur serially.

3.2. Time-sharing Conditions in the Intermediate Form

The time-sharing condition asserted over the execution space has to be transformed into some condition asserted over the generator space (intermediate form) so that time-sharing candidates can be identified by just examining the intermediate form of a program. It is assumed that only the data dependency within an assignment is exploited. That is, the orig-

inal control structure described in the given program is maintained. When the control structure is changed, either by removing some control sequencing originally described in the program or by augmenting additional control sequencing, some adjustments to the time-sharing condition stated below may have to be made.

The intermediate form of a program is obtained by repeatedly applying the rules in fig. 2(a)-(h) until a network of data and control transitions is obtained. Some terms are defined concerning the data transitions in the network obtained by these expansions. A data transition T is said to be from a process P if T is a data transition in the network obtained by expanding P according to the rules in fig. 2(a)-(h). Processes P_1 and P_2 are said to be composed by the SEQ(PAR) construct if the control dependency between them is as shown in fig. 2(d) and 2(e). Compositions involving predicates, e.g. IF, WHILE, ALT can be defined similarly. Two data transitions T_1 and T_2 are said to be *mutually exclusive* if T_1 and T_2 are, respectively, from P_1 and P_2 that are composed by the IF or ALT construct. Two data transitions T_1 and T_2 are said to be *sequential* if T_1 is from P_1, T_2 is from P_2, and P_1 and P_2 are composed by SEQ (fig. 2(a)). From this it is clear that T_1 and T_2 will not be enabled together at any of the reachable marking from the initial marking and after the firing of T_1 only T_2 can fire.

Two transitions are also said to be *sequential* if they belong to one of the following four classes: (1) if transition T_1 is from P_1 composed by a WHILE construct, T_2 is from the boolean condition Q of the WHILE construct (fig. 2(c)), (2) T_1 is from P_1 or P_2 which are composed by IF construct, T_2 is from the boolean condition Q (fig. 2(d)), (3) T_1 is from P_1 (P_2) and T_2 is from Q_1 (Q_2) within an ALT construct (fig. 2(e)), or (4) T_1 and T_2 are from an assignment statement or a predicate (e.g. Q in fig. 2(c)-(e)), and every output place of T_1 precedes every input place of T_2.

Theorem 1. ***(Time-Sharing Theorem)*** *Instances of two data transitions that are sequential or mutually exclusive can time-share a circuit.*

Proof: If data transitions T_1 and T_2 are sequential, then T_1 and T_2 will be either (1) from assignment (or predicate) statement or (2) from statement other than assignment (or predicate). In the former case, it is clear, from the flow relation in the net, that all tokens at the

output places of T_1 are consumed before a token is put at an input place of T_2. For a valid occam program, the assignment statement will not be enabled again until it is completed. Thus, all tokens at the output places of T_2 will be removed before any one of the input places of T_1 receives a token.

In the latter case, by definition, there exist processes P_1 and P_2 (or some predicate Q, but the proof will be similar) from which T_1 and T_2 are obtained such that P_1 and P_2 are composed by the SEQ control construct. (For the case that T_1 is from a process and T_2 is from a predicate, the control construct that composes them can be WHILE, IF or ALT.) Since all the data transitions in P_1 are fired before any one of the data transitions in P_2 is fired, and, for the intermediate form of any valid occam program, all the data transitions in P_2 are fired before any one of the data transitions in P_1 is fired again, it follows that the instances of T_1 and T_2 occur serially and so they can time-share a circuit.

If data transitions T_1 and T_2 are mutually exclusive, then, by definition, there exist processes P_1 and P_2 from which T_1 and T_2 are obtained such that P_1 and P_2 are composed by the IF or ALT construct. Since, for the intermediate form of any valid occam program, all the data transitions in P_1 are fired before the same instance of the control construct that composes P_1 and P_2 is enabled again (and hence any one of the data transitions in P_2 can have a chance to be fired), and vice versa, it follows that the instances of T_1 and T_2 occur serially and hence they can time-share a circuit. ■

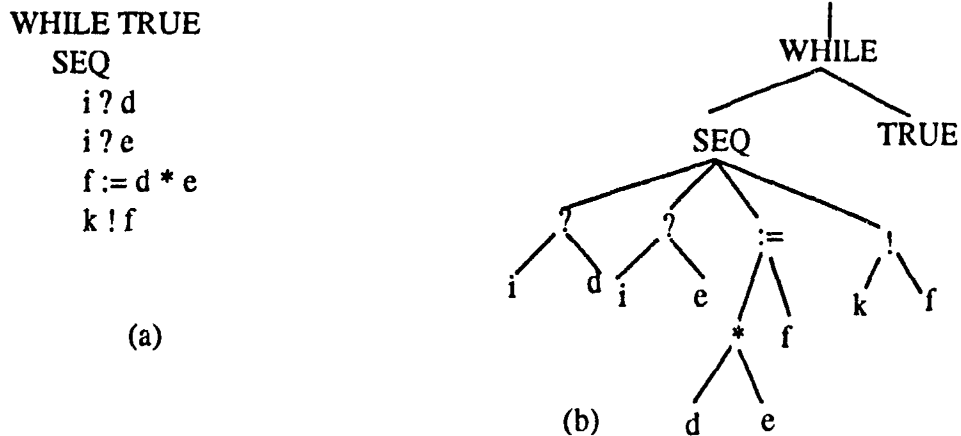
3.3. Procedure for Identifying Time-Sharing Candidates

Theorem 1 gives a condition which is to be checked against the intermediate form to identify time-sharing candidates. An abstract procedure is described below.

Given an occam program, a parse tree of the program can be constructed along with the compilation of the program into the intermediate form. A node of the parse tree is labeled with some control construct. The children of the node are labeled with the components that constitute the control construct. For example, for the WHILE construct, a node labeled WHILE in the parse tree has four children which are labeled M, Q, B and P_1. They correspond to the transitions in fig. 2(f). The node labeled Q will be the root of a sub-tree. The children of the node labeled Q are the data transitions as a result of expanding Q. The node labeled P_1 will be the root of a sub-tree for the process P_1. Suppose P_1 is the PAR composition of a set of processes, then the node labeled P_1 will only have a child labeled PAR.

Cross-references between data transitions in the intermediate form and the leaves of the parse tree are maintained.Consider the occam program in fig. 3.1(a). The parse tree for the occam program is given in fig. 3.1(b).

Fig. 3.1 An example occam program and its parse tree



Given any two data transitions T_1 and T_2, if they are from a predicate or an assignment statement, then whether they are sequential can easily be checked using the intermediate form. If they are from neither a predicate nor an assignment, then it is possible to get the smallest sub-tree of the parse tree such that T_1 and T_2 are leaves of the sub-tree, and the root of the sub-tree is labeled with the name of a control construct. The conditions stated in the definitions of sequential and mutually exclusive transitions can then be checked.

Correctness of the procedure is established as follows. If T_1 and T_2 are from a predicate or assignment statement, then it follows that the checking done in the procedure is just what is stated in the definition for sequential pair of data transitions. Otherwise, they are from some predicate or processes that are composed using one of the control constructs (as the intermediate form is obtained by recursively applying the rules in fig. 2(a)-(h)). That particular control construct is given by the label of the root of the smallest sub-tree mentioned in the procedure. Once it is identified, the rest are just checking whatever stated in the definition. IFORM thus meets the requirement on allowing the detection of transitions that can time-share a circuit.

3.4. Circuit for Realizing Time-Sharing

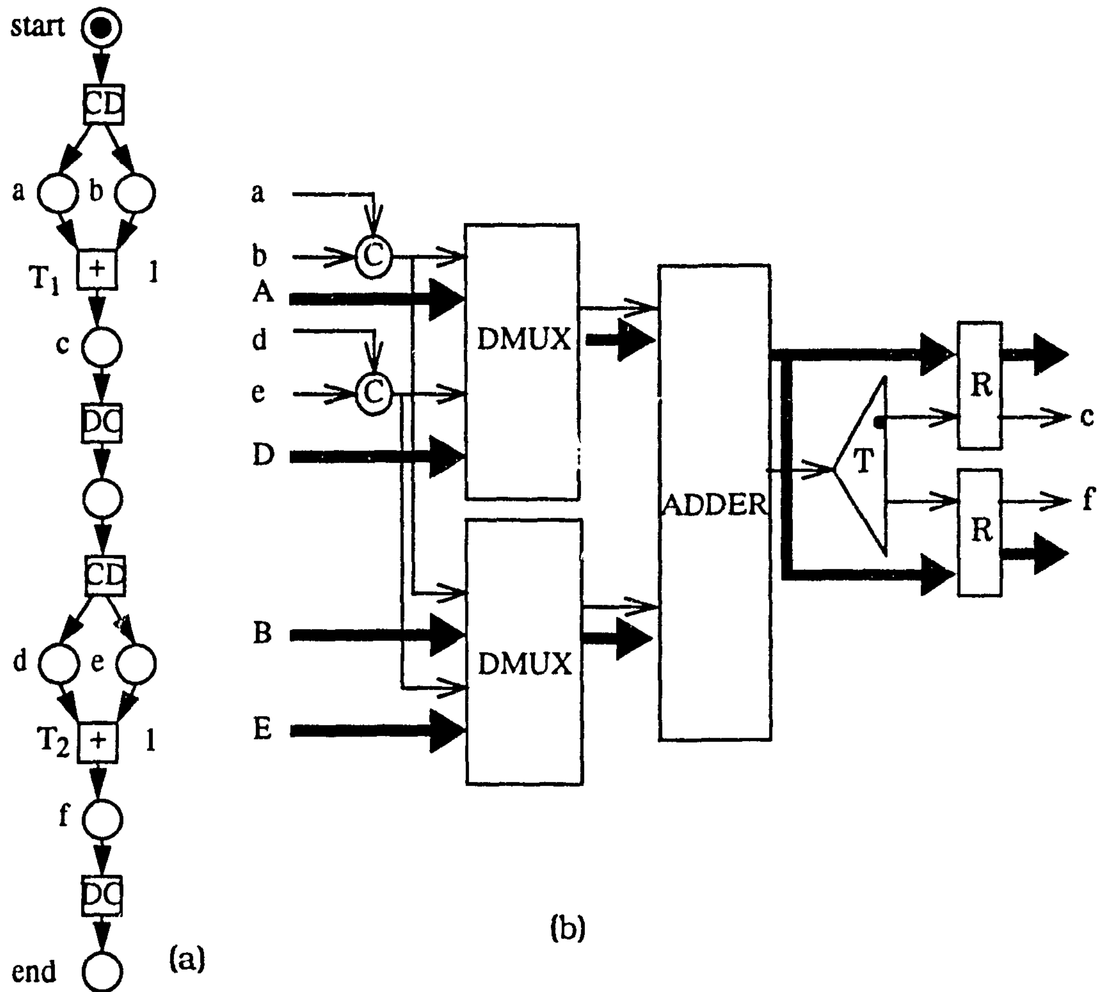
The circuit for routing data values to the physical circuit being time-shared is similar to the circuit required for routing data values (corresponding to output processes communicating over a channel) to the same physical channel. As a result, use of the CALL element in fig. 2(i) is still valid in this context.

Fig. 3.2(b) provides a circuit for realizing time-sharing corresponding to the IFORM in fig. 3.2(a). The transitions T_1 and T_2 are labeled with the same number 1 to denote that they have to be mapped into a single physical adder circuit. The C-elements make sure that the required data values a and b / d and e are available. The output of the C-elements are used to de-multiplex the data inputs. Once the data values have been de-multiplexed the addition operation is performed by an adder circuit module. The return transitions from the CALL module is used to latch the output data value from the adder module.

Fig. 3.2 Circuit for realizing Time-Sharing

If the transitions which are time-shared in the circuit implementation are fired alternatively at all times, then a simpler circuit could be provided. Fig. 3.3 provides a circuit for realizing time-sharing. It has a TOGGLE circuit module instead of a CALL module. The alternating output signal transitions from the TOGGLE module is used to latch the output data value from the adder in one of the registers alternatingly.

Fig. 3.3. Circuit for realizing Time-Sharing



Time-sharing of large data manipulating circuit modules will reduce the area significantly. Since the data transitions which are only sequential or mutually exclusive are labeled for time-sharing, the performance of the circuit does not reduce much. If area of the circuit has to be reduced further, then the IFORM could be modified in such a way more transitions could be chosen for time-sharing without violating the correctness criteria. The next chapter discusses this aspect further.

Chapter 4

Augmentation, Performance Evaluation, and Optimization Strategy

The intermediate form could be modified by adding or removing edges, transitions and places so that the performance of the derived circuit can be improved or the area of the circuit can be reduced without violating the correctness criteria. The firing rules of the transitions can also be changed so that concurrent transitions can be mapped to a single physical circuit by time-sharing. Sur'. modifications to an intermediate form are called augmentations. In this chapter augmentation by data flow ex' uon and by serialization are presented.

1. Augmentation by data flow extraction

The intermediate form could be augmented by removing some of the control sequencing in the original program. While performing sequential composition of IFORMs, data flow could be extracted. The extracted data flow can be used to merge directly the output data place of a transition (producer) with the input place of another transition (consumer) without the need for "DC-CD" and "CD-DC" transitions. Such augmentation also increases concurrency. However, control flow among the input/output operations must be maintained.

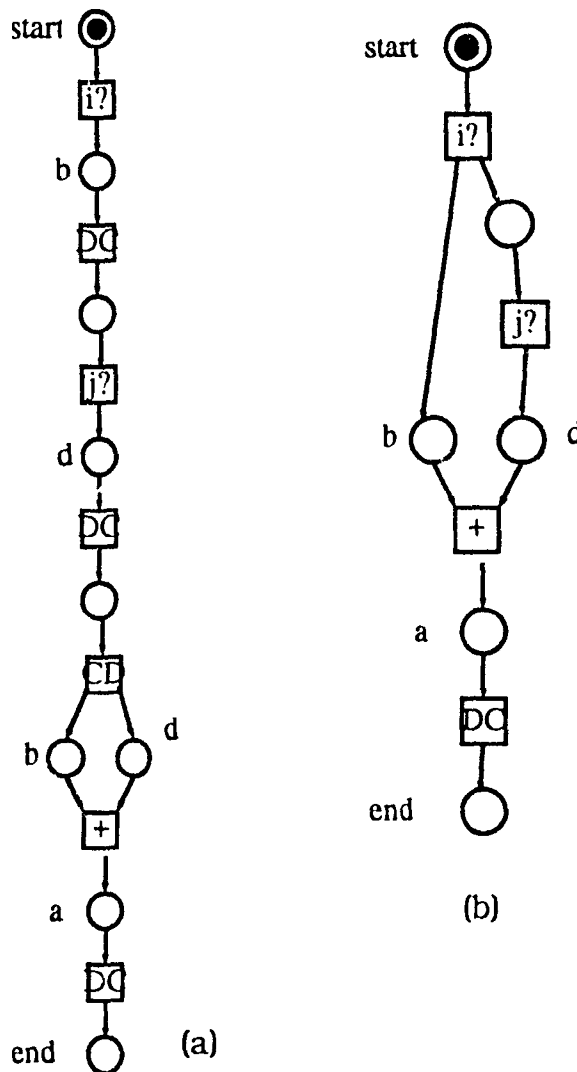
Consider the IFORMs (a) and (b) in Fig. 4.1. The IFORM (a) is augmented by data flow extraction to get IFORM (b). The output place of the i? is merged with the input place of the + transition. Similarly the output place of the j? transition is merged with the input place of the + transition. Such merging preserves the producer-consumer data dependency. To keep the control flow among the two input transitions, an additional output place is added to the i? transition which also forms the input place of the j? transition.

4.2. Augmentation by serialization

Instances of data transitions that are neither sequential nor mutually exclusive can not time-share a circuit in the physical implementation. If they are to time-share a circuit, then special provision will have to be made. Otherwise, new input data values will be given to the (data manipulating) circuit before the values produced are all consumed, and there will be a potential danger for data contamination. This section describes some augmentation strategies for modifying the intermediate form so that data transitions that are previously neither sequential nor mutually exclusive become so, and time-sharing can then be applied.

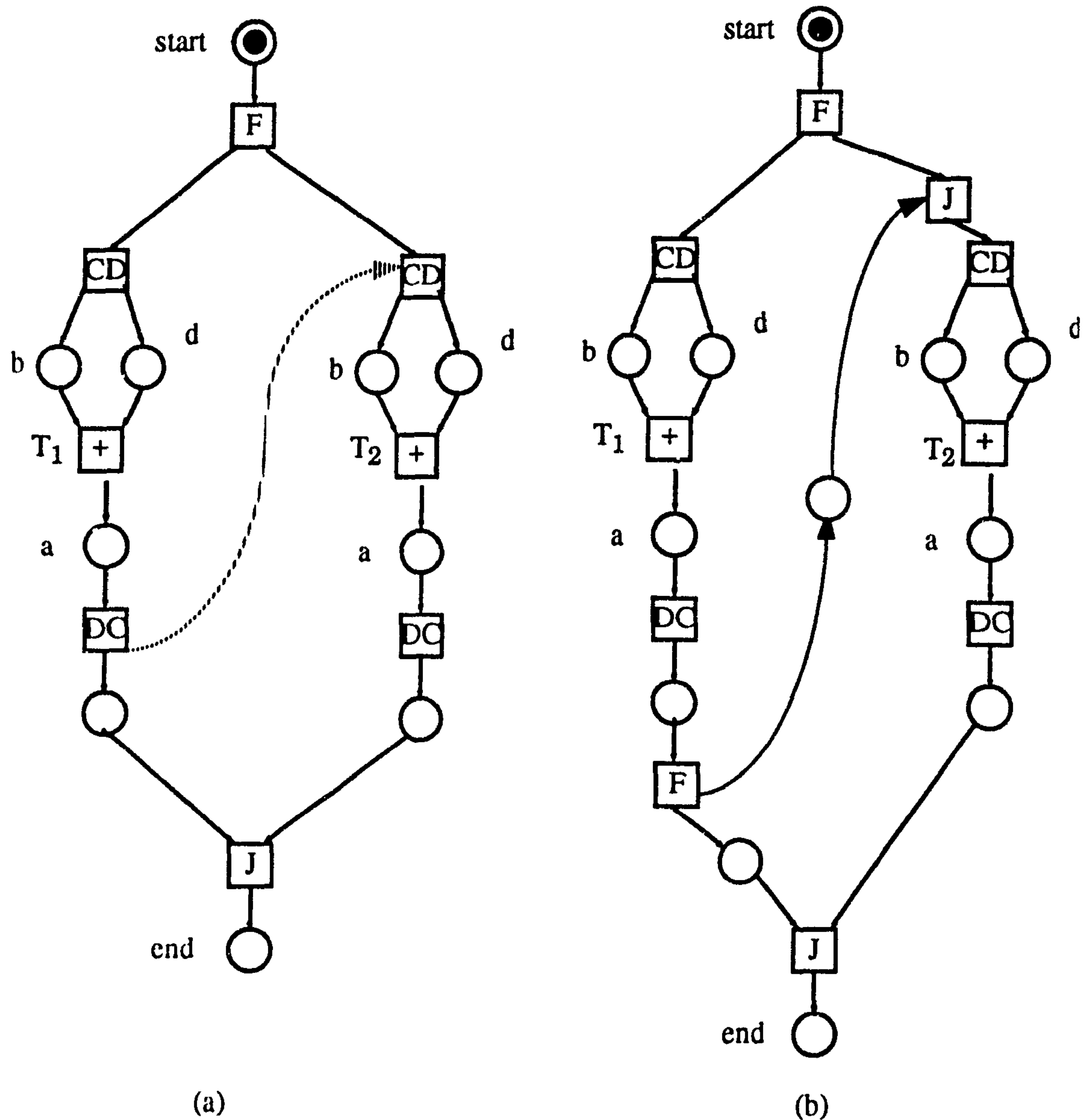
Consider the IFORM (a) in Fig. 4.2. The + transitions are neither sequential nor

Fig. 4.1 Augmentation by data flow extraction



mutually exclusive. By adding an edge from one of the transition to another forces the transitions to become sequential. The augmented IFORM (b) is then derived by including a J(JOIN) transition to join the two edges coming to the + transition.

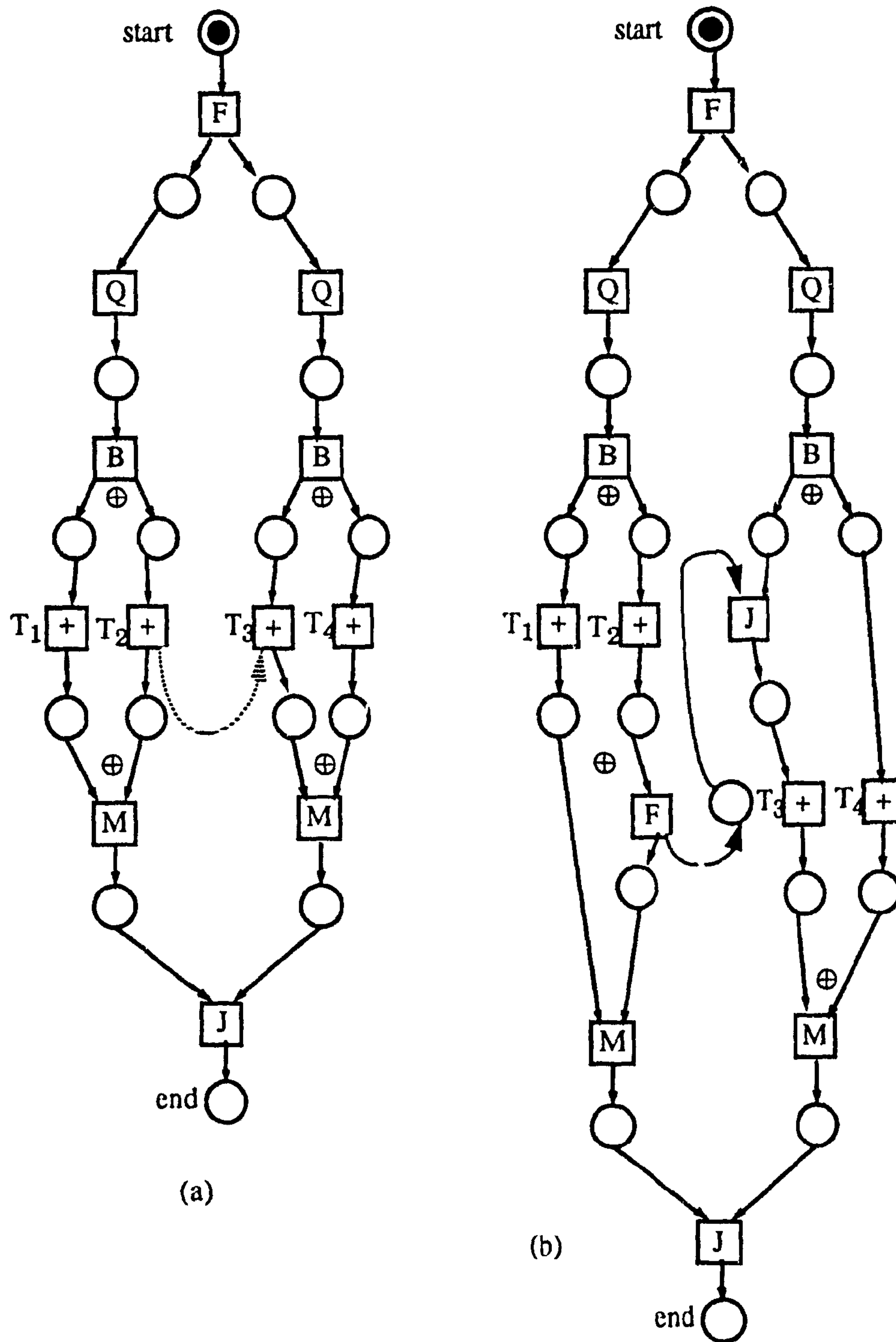
Fig. 4.2. Augmentation by serialization



The augmented IFORM is a correctness preserving implementation of the IFORM (a) in fig. 4.2. Observe that in both of the IFORMs a token will be put in the end place only

after the two + transitions have fired once. A proof of correctness is as follows: For each causal relation between any pair of data transitions in (b) there is a causal relation between corresponding pair of data transitions in (a). However, the reverse is not true. As a result, every possible execution sequence of data transitions in (b), the set of sequence of execu-

Fig. 4.3. Illegal augmentation

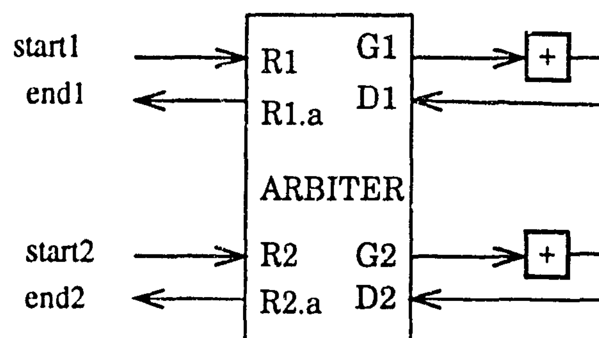


tion of data transitions is {T1T2}, while that in (a) is {T1T2, T2T1}. Thus, by definition, IFORM (b) realizes IFORM (a).

If unchecked, adding an edge between arbitrary transitions may result in progress violation. Consider the IFORM in Fig. 4.3. Transitions T_2 and T_3 are neither sequential nor mutually exclusive. By adding an edge between transitions T_2 and T_3 will result in progress violation if transitions T_1 and T_3 are enabled after the firing of the B(BRANCH) transitions. Thus the IFORM (b) in Fig 4.3 is an incorrect implementation of IFORM (a).

Transitions T_2 and T_3 can be serialized by labelling them with the same label. To differentiate this type of labelling from other types mentioned in this thesis, the label name is chosen from the greek alphabets. Transitions T2 and T3 can be labeled with the letter "α". The firing rules for the labeled transitions are modified as follows: whenever only one of the labeled transitions is enabled that transition will be fired. When all the labeled transitions are enabled only one of the transitions will be fired while all other transitions will still remain enabled. After the firing of the transition another transition is chosen arbi- ly for firing. Note that an enabled transition can be re-enabled only after it has fired. Thus the IFORM meets the requirement on allowing the detection of transitions between which augmentation can be performed. While translating an IFORM into circuit, transitions which are serialized by labeling can be mapped into a single physical circuit component along with the use of an arbiter. Fig. 4.4 gives a circuit for realizing time-sharing corresponding to transitions T2 and T3 in the IFORM (a) in Fig.4.3. A signal transition

Fig. 4.4. Circuit for realizing Time-sharing



can occur at start1 only after an acknowledge transition has been initiated at end1 for the previous, if any, start1 signal transition. The output signal transitions of the adders are used by the arbiter to initiate the acknowledge signal transitions at end1 and end2.

Augmenting an IFORM by serialization results in more time-sharable transitions. If the overhead of time-sharing is minimal comapared to using additional circuit module, then such augmentations will minimize the area of the resulting circuit. Due to serialization computations have to be performed one after the other and this reduce the performance of the circuit much.

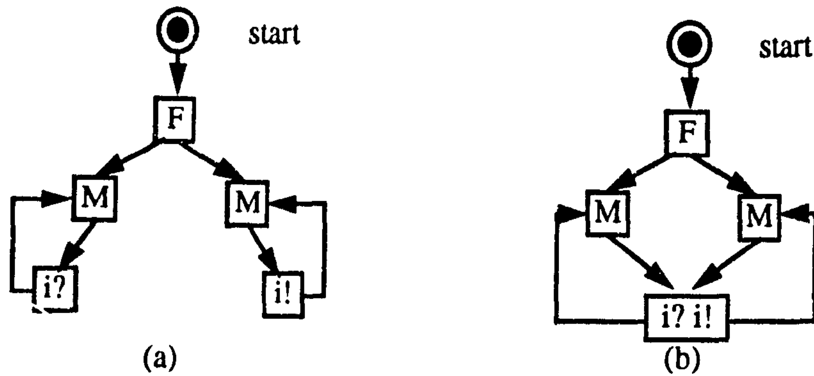
4.3. Performance Evaluation

A performance measure called cycle time for Petri nets has been discussed in [6, 22,]. Also, [22, 30] gave some methods for determining a performance measure for an event-rule system that models some deterministic computation. However, none of these are generally useful for evaluating the performance from an intermediate form. Furthermore, [6] did not provide methods for non-deterministic computations which are certainly present in any general occam program. The performance measures in [22, 30] assumed that there are no special firing rules in the given Petri net which is not the case for the intermediate form. The special firing rules for modeling arbitration and synchronous communication indicate that a different performance evaluation method is required.

For some special cases, e.g. the elliptic filter example in this thesis, the methods in [22, 30] are applicable. When the processes that communicate are deterministic, it is possible to adopt the performance measure for decision free Petri nets in [22, 30]. As in the elliptic filter example, the processes that communicate have nonterminating WHILE loops and there is only input/output process in each of the loops. Thus, it is possible to replace the two transitions corresponding to the input/output processes by a single transition in order to adopt the results in [22, 30]. Consider the IFORM (a) in Fig 4.5. Whenever the transition i? is

enabled it cannot fire until i! is also enabled. When both transitions are enabled they fire

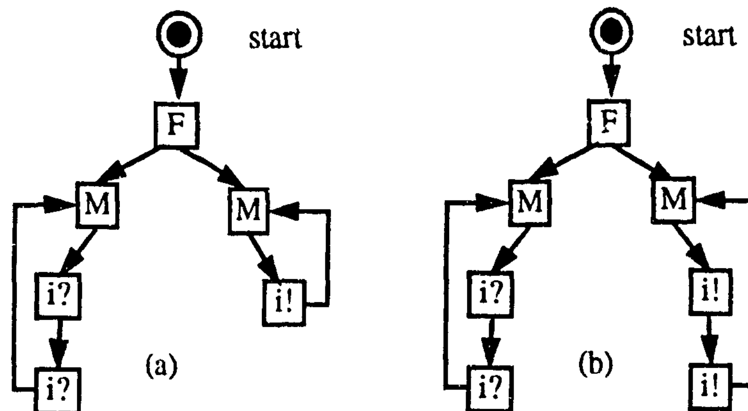
Fig. 4.5. Merging of Input/Output transitions



simultaneously. The two transitions can be merged into a single transition as in Fig 4.5(b) for the purpose of performance measurement.

In a more general case where the number of input processes communicating over a channel C in a nonterminating loop is different from the number of output processes communicating over C in another loop, the loops can be unfolded a finite number of times so that the numbers of input/output processes in the loops are identical. The transitions that represent the corresponding input/output processes can then be replaced by a single transition. Consider the IFORM in fig. 4.6(a). For one iteration of the loop (i? i? M), loop (i! M) iterates twice. The loop (i!, M) is unfolded as shown in fig. 4.6(b) so that each of the i? tran-

Fig. 4.6. Unfolding of loops for merging Input/Output transitions



sitions can be merged with a corresponding i! transition.

In general, the dynamic behaviors due to the interaction between non-determinism and synchronous communication can be quite complicated. Meaningful performance measures and methods for estimating them are to be explored.

4.4. Optimization Strategy

The optimization problem is defined as follows: Given an IFORM, a circuit implementation has to be derived such that it has minimum cost. The cost function could be given as the product of physical area of the circuit (A) and cycle time (T) [23]. Where A is less than or equal to a A_L, a constant.

Definitions:

In an IFORM, a sequence of places and transitions, $p_1\ t_1\ p_2\ t_2\ p_3\ t_3 \ldots p_n$ is a *directed path* from place p_1 to place p_n if transition t_i is both an output transition of place p_i and an input transition of place $p_{(i+1)}$ for $1 \leq i \leq n\text{-}1$. If p_1 equals p_n then the directed path is called *directed loop.*

If $c_1, c_2, \ldots, c_{n-1}$ are the circuit modules corresponding to transitions $t_1, t_2, \ldots, t_{n-1}$ in a directed loop, then the sum of delay times of the circuit modules $c_1, c_2, \ldots c_{n-1}$ is called the *loop time.*

Algorithm:

step1: Find all directed loops in the IFORM and find their loop times by adding the firing times of the transitions in a cycle. Order the loops in the ascending order of their loop times. Initially none of the transitions are labeled for time-sharing.

Following iterations are performed one after another. At any stage if the area is less than or equal to A_L, or if all the iterations have been done, the algorithm terminates.

Iteration-1: Sequential data transitions from a loop are chosen for time-sharing in such a way that these transitions are not present in the loop with maximum loop time. After labeling such transitions for time-sharing the loops containing those transitions should not become critical. A critical loop is one whose loop time is the maximum compared to other loops in the IFORM.

Iteration-2: Mutually-exclusive data transitions from different loops are chosen for time-sharing in such a way that these transitions are not present in the loop with maximum loop time. After labeling such transitions for time-sharing the loops containing those transitions should not become critical.

step-2: Transitions are selected for time-sharing which are neither sequential nor mutually exclusive. The transitions are augmented by any of the appropriate techniques described in sections 4.1 and 4.2. No new critical loops should result by doing such augmentations. The loops have to be extracted after performing the augmentation. Iterations 1 and 2 are performed again.

Iteration-3: Iterations 1 and 2 are repeated but with relaxed criteria. Transitions selected for time-sharing can be present in the critical loop or even if an non-critical cycle becomes critical after time-sharing.

The stated optimization method is applied in the synthesis of a fifth order wave elliptic filter. The results obtained are presented in chapter 5. A software implementation of the synthesis methodology presented in this thesis has been developed. Chapter 5 presents details of the software.

Chapter 5

Software implementation

A synthesis software (ODIT) has been developed in C++ under SUNOS 4.1 environment. The software accepts an occam specification of a system and generates the intermediate form (IFORM). The software provides an user interface for the user to provide the number of data manipulating circuit modules to be used in the synthesis. The software performs optimization based on this criteria and generates the netlist in Electronic Data Interchange Format (EDIF). The user is also provided with the estimated performance of the circuit. As an option the software can be run to get a circuit by syntax-directed translation.

The software has the following modules:

1. Parse tree constructor

The module which takes the occam program as its input is the parser module. The parser has been written using the yacc unix programming tool. The grammar rules of occam and actions to perform when a syntactic entity/construct is recognized in the input programs are provided as input for yacc to generate the parser. The parser generator module is invoked in the main function by calling yyparse() function. The yyparse() function which is generated by the yacc tool, will in turn call yylex(). The yylex() function when called will return the next token in the input. The actions performed when grammar rules are recognized include symbol table creation and parse tree construction. While constructing the parse tree, the producer of each data value is also stored with the symbol reference. The module will pass the head of the parse tree structure to the next module for further processing.

2. Parse tree to IFORM translator module

This module takes the head node of parse tree as its input and generates another data structure which is the IFORM. While constructing the IFORM the already stored producer-consumer relations are used in merging the output place of a pro-

ducer with the input place of a consumer.

The algorithm to construct IFORM from parse tree consists of two steps. The first step involves in finding the producers for the data values used with in a composition constructs IF, ALT and WHILE constructs. The second step builds the IFORM structure. The first step is as follows:

```
void scan_tnode( node )
/* function is called with input argument pointing to the head of the parse tree. */
{
 if( node != NULL )
 {      switch( node type ){
        case T_PROC: /* Process composition */
               scan_tnode( proc_process_list );
        case T_INPUT:
               scan_tnode( next_node );
        case T_OUTPUT:
               scan_tnode( next_node);
        case T_ASSIGN:
               scan_tnode( next_node );
        case T_SEQ:
               scan_tnode( seq_process_list);
               scan_tnode( next_node);
        case T_PAR:
               scan_tnode( par_process_list);
               scan_tnode( next_node );
        case T_IF:
               get all consumers and producers (variables) in both blocks of IF
               construct;
        case T_ALT:
```

```
            get all consumers and producers (variables) in both blocks of ALT
            construct;
        case T_WHILE:
            get all consumers and producers in the WHILE block }
}
```

The second step is as follows:

```
void trav_tnode( node ){
if( node != NULL ){
        switch( node type ){
        case T_PROC:
            trav_tnode(proc_process_list );
        case T_INPUT:
            create input transition;
            assign the start event for the transition;
            update the current producer's list and start event list;
            set the input transition pointer for the channel symbol to transition;
            trav_tnode(next_node);
        case T_OUTPUT:
            create output transition;
            assign the start event and producer for the transition;
            update the current start event list;
            set the input transition pointer for the channel symbol to transition;
            trav_tnode(next_node);
        case T_ASSIGN:
            create data transition of operator type;
            assign the producers for the data transition;
            update the current producer's list;
```

```
        trav_tnode(next_node);
    case T_SEQ:
        trav_tnode(seqlist );
        trav_tnode(next_node);
        break;
    case T_PAR:
        create fork transition;
        create join transition;
        while(par_process_list_next_node_not_empty(){
            set start event = fork transition;
            trav_tnode(cur_node_in_par_process_list);
            insert current start event to the list of start events for join;
            next_node_in_par_process_list;}
    case T_WHILE:
        while(!is_consumers_list_empty()){
            /* Consumers are the variables which has data dependency
            with variables used before the while block *
            get consumer();
            create data_join and data_merge transitions();
            assign producer for data values for data_join;
            insert data_join and data_merge in the IFORM list;
            update the data value producer and the current start event
            producer;
            nextnode_in_consumer's_list(); }
        trav_tnode(process_list);
        assign producer for data values for data_merge_transitions;
        trav_tnode(next_node);
    case T_IF:
        while(!is_consumers_list_empty()){
```

```
            create a branch tranition and merge transition;
            assign the producer and start transitions for branch;
        }
        trav_tnode(if_branch_node);
        set start transition to branch transition;
        trav_tnode(else_branch_node);
        trav_tnode(next_node);}
    case T_ALT:
        create alt_fork transition;
        create the data join transitions;
        create transitions for conditionals;
        set entry transition;
        trav_tnode(guarded_block);
        set start transition to alt_fork;
        trav_tnode(skip_block);
        trav_tnode(next_node);}}}
```

3. Module to get all simple loops in IFORM:

/* The input and output operations on the same channel are merged together in the following way: In the symbol table for channel names, pointers for the input and output process is maintained. Whenever an input object is reached while traversing the IFORM, the child nodes of a complementary input/output process is also taken as the child nodes of the current node. All the loops of the IFORM are extracted. For each of the loop there is a linked list of transitions (objects). The transitions are ordered in their order of firing in the IFORM. */

```
while(next_channel_in_symbol_table){
    if(input and output transitions are set){
        append all consumer transitions of input transition to the consumer
        transition list of output transition;
```

```
        append all consumer transitions of output transition to the consumer
        transition list of input transition;}}

traverse(next_node_in_IFORM()){
    check_it_was_in_the_current_loop;
    if(in current loop){
        check the loop is a duplicate loop;
        if(not duplicate loop){
            add in to locps list;}}
    else  {
        traverse( consumers);}}
```

4. Module to sort the list of loops:

/* The loop times of each of the loop is calculated. To calculate such a figure the operator.h file will have #define entries for each of the basic circuit component. Each of the defined names corresponds to the average delay times of the circuit components. The loops are sorted in the ascending order of their loop time. */

```
while(next loop){
    calculate loop time;}
sort loops in the ascending order of loop times;
```

5. Optimizer module:

/* The optimization function will try to meet the user's requirement regarding the number of data manipulating circuit components to be used in synthesis. For example user will provide "-mul 4" as an option while executing the odit program. If the number of multiplications operations are more than 4, then the optimize function has to find sufficient number of transitions which could be labeled for time-sharing in such a way that the number of required circuit component satis-

fies user's requirement. */

The optimization is done in the following steps:

```
step1:
 while(next loop in the sorted list of loops){
         while(next transition in loop){
                 if(transition type == CURRENT OPTIMIZE OPERATOR TYPE){
                         calculate the loops times assuming the transition is labelled;
                         if(! any new critical loop ){
                                 label transition for time-sharing;}}}}
 set to first loop;
 step 2:
 while(next loop in the sorted list of loops){
         while(next transition in loop){
                 if(transition type == CURRENT OPTIMIZE OPERATOR TYPE
                         and not labelled for time-sharing){
                         get similar type of transition from another cycle which is
                          not labeled and which is mutually exclusive;
                         calculate the loop times assuming the transition is labelled;}
                 if(! any new critical loop ){
                         label transitions for time-sharing;}}}}
set to first loop;
step 3:
while(next loop in the sorted list of loops){
         while(next transition in loop){
                 if(transition type == CURRENT OPTIMIZE OPERATOR TYPE
                         and not labelled for time-sharing){
                         get similar type of transition from another cycle which is
                          not labeled and which is concurrent;
                         calculate the loop times assuming the transition is labelled;}
```

```
        if(! any new critical loop ){
            augment the transitions so that the transitions become
            sequential;
            label transitions for time-sharing;}}}}
repeat steps 1, 2, and 3 without checking for new critical loop;
```

6. IFORM to netlist translator module:

```
/* The IFORM is traversed and for each of the transition, according to the type of
transition a EDIF netlist template is selected. The current instance number of basic
circuit components used and the transitions number of consumer transitions are
used to generate the netlist. */
traverse(next node in IFORM){
    increment component number for the corresponding type of component;
    get consumer transition numbers;
    select EDIF template for the current transition;
    if( transition is labeled){
        include template for time-sharing circuit;
    generate netlist by including the transition numbers;
    print the netlist;
    traverse(next node);}
```

Support functions:

1. Function to find transition pair type:

```
/*This function takes two transitions as its input and returns the transition type
(sequential or concurrent or mutually exclusive or conditionally concurrent).*/
pair_type find_pair_type(transition1, transition2){
    while(next cycle){
        while(next transition){
            if(transition = transition1){
```

```
                check if transition2 is found in the rest of the cycle;
                if (found){
                        return sequential;
find_all_least_common_ancestors(transitions1, transition2);
if(all l.c.a are fork transitions){
        if(any decisition transtion found )
                return conditionally concurrent;
        else
                return concurrent;
return mutually exclusive;
```

When the synthesis program "odit" is invoked, it will read the occam specifica- .. through standard input and output the EDIF netlist of the circuit synthesized by the syntax directed translation. The program will then list the number of data manipulating circuit modules like adders, multipliers, etc. used in the syntax directed translation. The user is asked to provide the number of circuit modules to be used for each type of data manipulating circuit modules. According to the user's requirement, the program will perform optimization to choose a circuit implementation which performs better and outputs the EDIF netlist of the synthesized circuit. The program also accepts command line option "-s" to perform no time-sharing optimization and to generate EDIF netlist of circuit synthesized by syntax directed translation.

The EDIF netlist generated by the synthesis program can be converted to CADENCE design environment by the "edifin" utility provided by the CADENCE VLSI CAD system. The cells corresponding to the basic circuit modules used by the synthesis program have to be present in one of the system directories for the conversion program to work. The CAD system can be used to do automatic place and route and generate the final layout.

5.1. The Elliptic Filter Example

A fifth order wave elliptic filter [14] is used as a demonstration of the proven as well as the conjectured results in chapter 4. The occam specification of the elliptic filter is shown in fig. 5.1. The synthesis software has been used in deriving various circuit netlists according to different optimization criteria. The IFORM for the elliptic filter example is shown in fig. 5.2. The places and "CD" and "DC" transitions are not shown in the figure for simplicity.

Manual synthesis of the elliptic filter has been performed using the synthesis methodology presented in this thesis. Cadence's SDA software has been used in performing automatic place and route to get the circuit layout. While performing routing care is taken to ensure the control and data wires are kept parallel to each other. The circuit modules are also designed in such a way that sufficient delay has been introduced in the control path to compensate the delay in computing circuit. DI cells developed by the VLSI research group in Concordia University and standard cells from CMC have been used in the design. The [illegible]s were designed using Northern Telecom's CMOS4S 1.2 μ technology.

The layout for 4 multiplier implementation of the elliptic filter is shown in fig. 5.3. Table 1 gives the results obtained for three different implementations of the elliptic filter. The overhead of time-sharing involves additional registers, and multiplexers. Multiplier circuit modules are the only circuit module found to be helpful in minimizing the area when time-shared. The results presented in the literature for the synchronous systems show the number of cycles taken for different number of multipliers and adders used in the

Table 1: Synthesis results

Multipliers	Area (μ^2)	Cycle time (nsec)
8	51,572,432	812
4	37,360,211	956
2	30,982,359	1172

synthesis. It is assumed that adders take one clock cycle and multipliers take two clock cycles. Since there is no clock in asynchronous circuits, it is not possible to present our

results in terms of clock cycles. Table 2 gives the results obtained using HAL [28] synthesis tool.

Table 2: HAL synthesis results

No of cycles	Multipliers	Adders
17	3	3
18	2	3
19	2	2
20	2	2
21	1	2

Fig. 5.1. Occam specification of an Elliptic filter

```
PROC ELLIPTICFILTER(in, out)
    VAR z1,z2,z3,z4,z5,z6,z7,x,x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,x11,x12,x13,
       x14,x15,x16,x17,x18,x19,x20,x21,x22,x23,x24,x25,x26,x27,
       y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11
       CHAN in, out, a, b, c, d, e, f, g, h, i, j, k
    PAR
       WHILE TRUE
          SEQ
             PAR
                SEQ
                   PAR
                      SEQ
                         PAR
                            SEQ
                               in? x
                               x1 := z1 + x
                            a? y1
                         x2 := x1 + y1
                      b? y2
                   x3 := x2 + y2
                c? y3
             x4 := x3 + y3
          PAR
             d! x4
             e! x4
             SEQ
```

```
          x5 := x4 * x4
          x6 := x2 + x5
          PAR
            f! x6
            g! x6
            SEQ
              x7 := x6 + x2
              x8 := x7 * x7
              x9 := x1 + x8
              PAR
                h! x9
                SEQ
                  x10 := x1 + x9
                  x11 := x10 * x10
                  x12 := x + x11
                  z1 := x12 + x9
WHILE TRUE
  PAR
    SEQ
      PAR
        a! z2
        b! z3
        SEQ
          PAR
            SEQ
              PAR
                f? y4
                e? y5
              x13 := y4 + y5
            i? y6
          z3 := x13 + y6
    SEQ
      PAR
        g? y7
        h? y8
      x14 := y7 + y8
      x15 := x14 + z4
      x16 := x15 * x15
      z4 := x16 + z4
      z2 := z4 + x14
WHILE TRUE
  SEQ
    PAR
```

```
            SEQ
                j? y9
                x17 := y9 + z5
                c! x17
            SEQ
                d? y10
                x18 := y10 * y10
        x19 := x17 + x18
        PAR
            i! x19
            SEQ
                x20 := x19 + x17
                x21 := x20 * x20
                x22 := x21 + y9
                PAR
                    k! x22
                    SEQ
                        x23 := x19 + x22
                        x24 := x23 + z6
                        x25 := x24 * x24
                        z6 := x25 + z6
                        z5 := z6 + x24
WHILE TRUE
    PAR
        j! z7
        SEQ
            k? y11
            x26 := z7 + y11
            x27 := x26 * x26
            PAR
                out! x27
                z7 := x27 + y11
```

Fig. 5.2. The IFORM for the elliptic filter

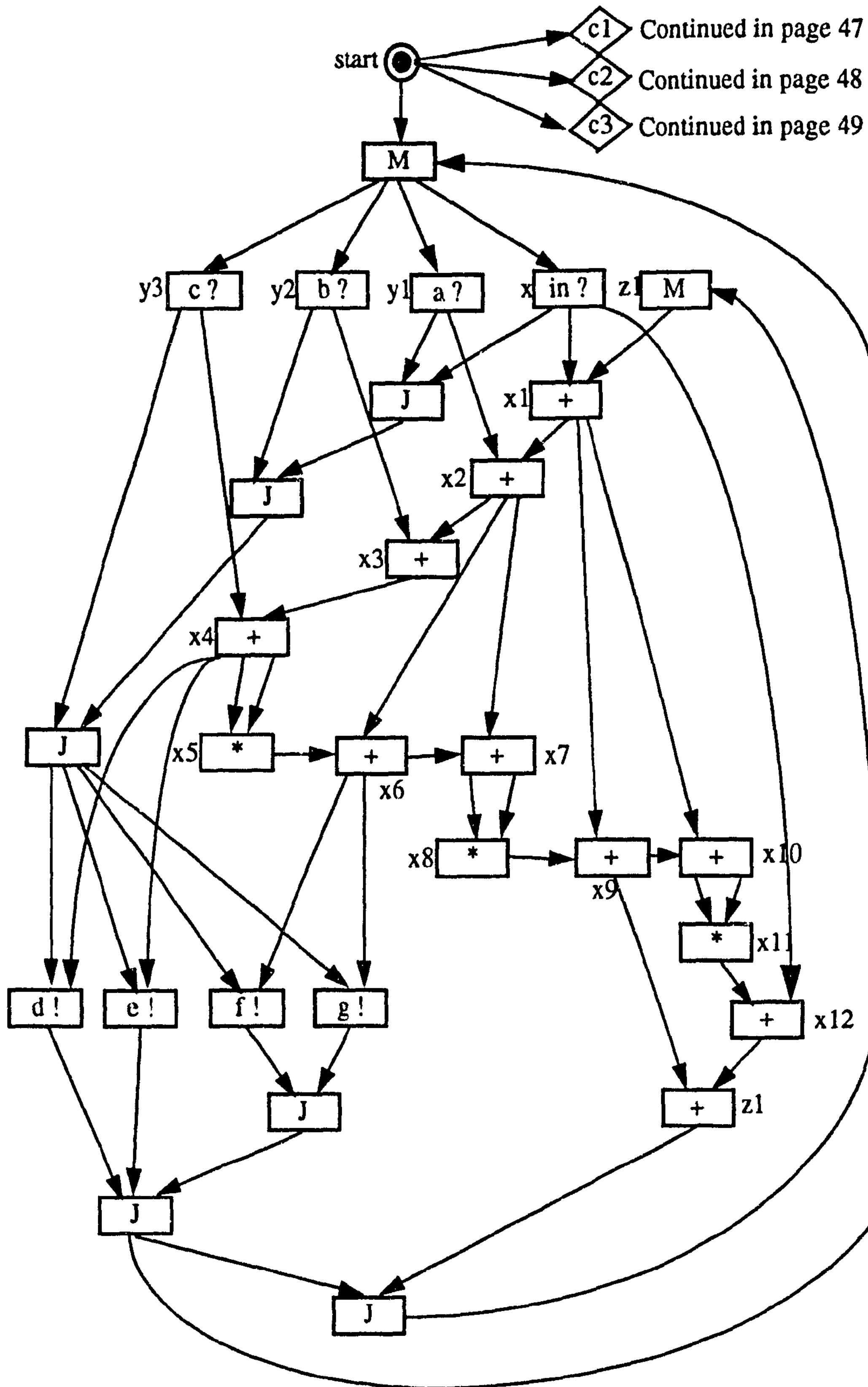


Fig. 5.2. The IFORM for the elliptic filter (continued)

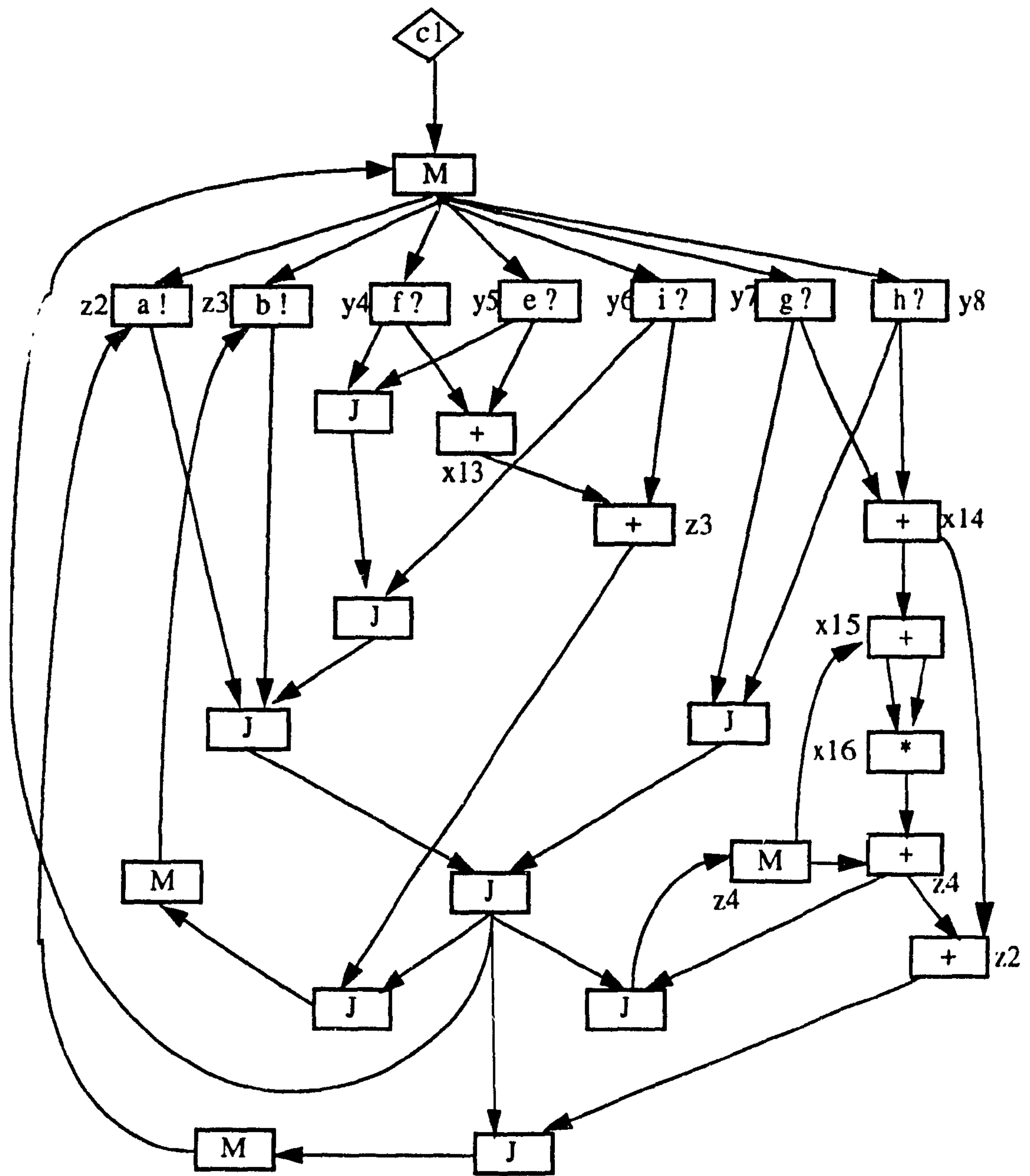


Fig. 5.2. The IFORM for the elliptic filter (continued)

Fig. 5.2. The IFORM for the elliptic filter (continued)

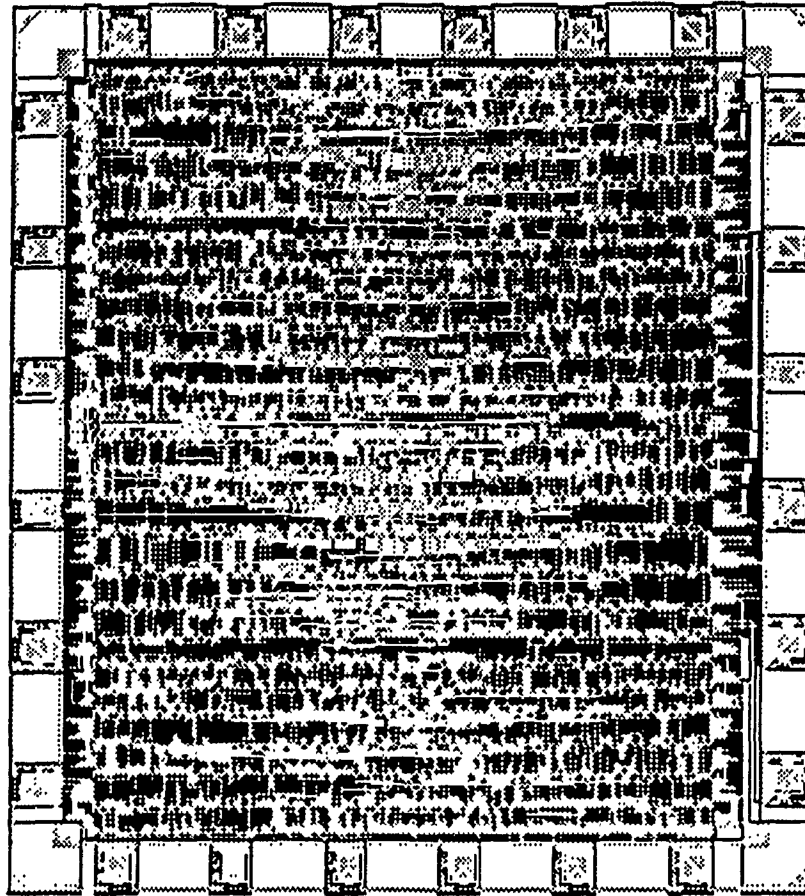
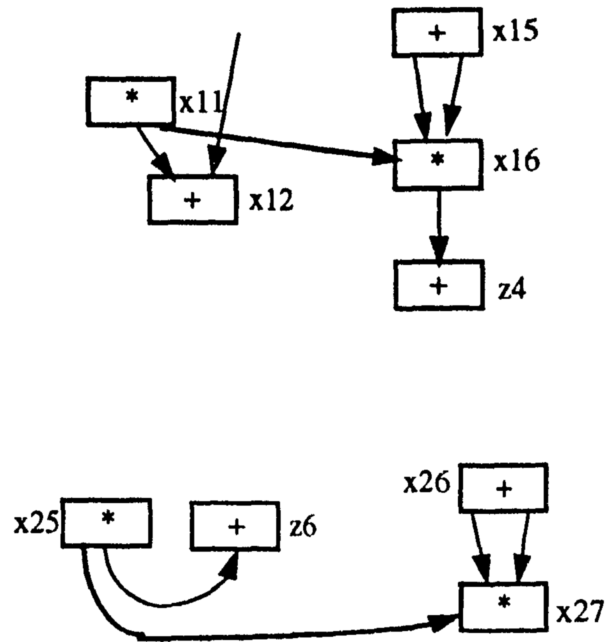


Fig.5.3. The layout of the optimized elliptic filter

The multiplier data transitions labeled by x5, x8 and x11 (fig. 5.3. page 46) are sequential and so can be labeled for time-sharing in the circuit implementation. Similarly multiplication transitions labeled x18, x21, x25 are also time-sharable as they are sequential. After labeling the above transitions for time-sharing, we require 4 multiplier circuit modules to implement the elliptic filter. To reduce the area further, the multiplier transitions labeled x11 and x16, and the multiplier transitions labeled x25 and x27 could be aug-

mented as shown in fig. 5.4. After augmenting transitions labeled x5, x8, x11 and x16 can

Fig. 5.4. Augmenting the IFORM by serialization



be labeled for timesharing. Similarly transitions labeled x18, x21, x25 and x27 can be labeled for timesharing. Only two multiplier circuit blocks are required to implement the augmented IFORM.

Further reduction in area could be achieved by timesharing a single multiplier circuit block by using an arbiter as explained in section 4.2. The EDIF netlist for the 2 multiplier implementation is given in fig. 5.5.

Fig. 5.5. EDIF Netlist of the circuit with four multiplier modules

```
(net start (joined (portRef start)
    (portRef OUT (instanceRef start_inpad))
    (portRef A (instanceRef xor2_1))
    (portRef A (instanceRef xor2_2))
    (portRef A (instanceRef xor2_3))
    (portRef A (instanceRef xor2_4))))
(net in (joined (portRef in)
    (portRef OUT (instanceRef in_inpad))
    (portRef A (instanceRef celement2_1))))
(net n1 (joined (portRef OUT (instanceRef xor2_1))
    (portRef A (instanceRef celement2_1))))
(net n2 (joined (portRef OUT (instanceRef celement2_1))
    (portRef A (instanceRef register_1))))
(net n3 (joined (portRef OUT (instanceRef register_1))
    (portRef A (instanceRef adder_1))))
(net n4 (joined (portRef OUT (instanceRef register_1))
    (portRef B (instanceRef celement2_1))))
(net n5 (joined (portRef OUT (instanceRef register_1))
    (portRef B (instanceRef adder_12))))
(net n6 (joined (portRef OUT (instanceRef xor2_5))
    (portRef B (instanceRef adder_1))))
(net n7 (joined (portRef OUT (instanceRef xor2_1))
    (portRef A (instanceRef celement2_2))))
(net n8 (joined (portRef OUT (instanceRef celement2_2))
    (portRef A (instanceRef register_2))))
(net n9 (joined (portRef OUT (instanceRef register_2))
    (portRef A (instanceRef adder_2))))
(net n10 (joined (portRef OUT (instanceRef register_2))
    (portRef A (instanceRef celement2_1))))
(net n11 (joined (portRef OUT (instanceRef xor2_1))
    (portRef A (instanceRef celement2_3))))
(net n12 (joined (portRef OUT (instanceRef celement2_3))
    (portRef A (instanceRef register_3))))
(net n13 (joined (portRef OUT (instanceRef register_3))
    (portRef A (instanceRef adder_3))))
(net n14 (joined (portRef OUT (instanceRef register_3))
    (portRef A (instanceRef celement2_2))))
(net n15 (joined (portRef OUT (instanceRef xor2_1))
    (portRef A (instanceRef celement2_4))))
(net n16 (joined (portRef OUT (instanceRef celement2_4))
    (portRef A (instanceRef register_4))))
(net n17 (joined (portRef OUT (instanceRef register_4))
    (portRef A (instanceRef adder_4))))
(net n18 (joined (portRef OUT (instanceRef register_4))
    (portRef A (instanceRef celement2_3))))
(net n19 (joined (portRef OUT (instanceRef celement2_1))
    (portRef B (instanceRef celement2_2))))
(net n20 (joined (portRef OUT (instanceRef celement2_2))
    (portRef B (instanceRef celement2_3))))
(net n21 (joined (portRef OUT (instanceRef adder_2))
    (portRef B (instanceRef adder_3))))
(net n22 (joined (portRef OUT (instanceRef adder_3))
    (portRef B (instanceRef adder_4))))
(net n23 (joined (portRef OUT (instanceRef adder_4))
    (portRef A (instanceRef celement2_9))))
(net n24 (joined (portRef OUT (instanceRef adder_4))
    (portRef B (instanceRef celement2_9))))
(net n25 (joined (portRef OUT (instanceRef celement2_3))
    (portRef A (instanceRef celement2_5))))
(net n26 (joined (portRef OUT (instanceRef celement2_3))
    (portRef A (instanceRef celement2_6))))
(net n27 (joined (portRef OUT (instanceRef celement2_3))
    (portRef A (instanceRef celement2_7))))
(net n28 (joined (portRef OUT (instanceRef celement2_3))
    (portRef A (instanceRef celement2_8))))
(net n29 (joined (portRef OUT (instanceRef adder_4))
    (portRef B (instanceRef celement2_5))))
(net n30 (joined (portRef OUT (instanceRef adder_4))
```

```
    (portRef B (instanceRef celement2_6))))
(net n31 (joined (portRef OUT (instanceRef register_5))
    (portRef B (instanceRef adder_5))))
(net n32 (joined (portRef OUT (instanceRef adder_2))
    (portRef A (instanceRef adder_5))))
(net n33 (joined (portRef OUT (instanceRef adder_2))
    (portRef A (instanceRef adder_6))))
(net n34 (joined (portRef OUT (instanceRef adder_5))
    (portRef B (instanceRef adder_6))))
(net n35 (joined (portRef OUT (instanceRef adder_6))
    (portRef A (instanceRef celement2_10))))
(net n36 (joined (portRef OUT (instanceRef adder_6))
    (portRef B (instanceRef celement2_10))))
(net n37 (joined (portRef OUT (instanceRef register_5))
    (portRef B (instanceRef adder_7))))
(net n38 (joined (portRef OUT (instanceRef adder_1))
    (portRef A (instanceRef adder_7))))
(net n39 (joined (portRef OUT (instanceRef adder_7))
    (portRef B (instanceRef adder_8))))
(net n40 (joined (portRef OUT (instanceRef adder_1))
    (portRef A (instanceRef adder_8))))
(net n41 (joined (portRef OUT (instanceRef adder_8))
    (portRef A (instanceRef celement2_11))))
(net n42 (joined (portRef OUT (instanceRef adder_8))
    (portRef B (instanceRef celement2_11))))
(net n43 (joined (portRef OUT (instanceRef register_6))
    (portRef A (instanceRef adder_9))))
(net n44 (joined (portRef OUT (instanceRef adder_7))
    (portRef A (instanceRef adder_10))))
(net n45 (joined (portRef OUT (instanceRef adder_9))
    (portRef B (instanceRef adder_10))))
(net n46 (joined (portRef OUT (instanceRef celement2_7))
    (portRef A (instanceRef celement2_4))))
(net n47 (joined (portRef OUT (instanceRef celement2_. ))
    (portRef B (instanceRef celement2_4))))
(net n48 (joined (portRef OUT (instanceRef celement2_5))
    (portRef A (instanceRef celement3_5))))
(net n49 (joined (portRef OUT (instanceRef celement2_6))
    (portRef B (instanceRef celement3_5))))
(net n50 (joined (portRef OUT (instanceRef celement2_1))
    (portRef C (instanceRef celement3_5))))
(net n51 (joined (portRef OUT (instanceRef celement3_5))
    (portRef A (instanceRef celement2_6))))
(net n52 (joined (portRef OUT (instanceRef adder_10))
    (portRef B (instanceRef celement2_6))))
(net n53 (joined (portRef OUT (instanceRef celement2_6))
    (portRef B (instanceRef xor2_2))))
(net n54 (joined (portRef OUT (instanceRef celement2_5))
    (portRef B (instanceRef xor2_1))))
(net n55 (joined (portRef OUT (instanceRef celement2_9))
    (portRef A (instanceRef dmux3_1))))
(net n56 (joined (portRef OUT (instanceRef celement2_10))
    (portRef B (instanceRef dmux3_1))))
(net n57 (joined (portRef OUT (instanceRef celement2_11))
    (portRef C (instanceRef dmux3_1))))
(net n58 (joined (portRef OUT1 (instanceRef dmux3_1))
    (portRef A (instanceRef multiplier_1))))
(net n59 (joined (portRef OUT2 (instanceRef dmux3_1))
    (portRef B (instanceRef multiplier_1))))
(net n60 (joined (portRef OUT (instanceRef multiplier_1))
    (portRef B (instanceRef toggle3_1))))
(net n61 (joined (portRef OUT (instanceRef toggle3_1))
    (portRef A (instanceRef register_5))))
(net n62 (joined (portRef OUT (instanceRef toggle3_1))
    (portRef A (instanceRef register_6))))
(net n63 (joined (portRef OUT (instanceRef toggle3_1))
    (portRef A (instanceRef register_7))))
(net n64 (joined (portRef OUT (instanceRef xor2_2))
    (portRef A (instanceRef celement2_12))
    (portRef A (instanceRef celement2_13))
    (portRef A (instanceRef celement2_14))
    (portRef A (instanceRef celement2_15))
```

```
      (portRef B (instanceRef celement2_6))))
    (net n31 (joined (portRef OUT (instanceRef register_5))
      (portRef B (instanceRef adder_5))))
    (net n32 (joined (portRef OUT (instanceRef adder_2))
      (portRef A (instanceRef adder_5))))
    (net n33 (joined (portRef OUT (instanceRef adder_2))
      (portRef A (instanceRef adder_6))))
    (net n34 (joined (portRef OUT (instanceRef adder_5))
      (portRef B (instanceRef adder_6))))
    (net n35 (joined (portRef OUT (instanceRef adder_6))
      (portRef A (instanceRef celement2_10))))
    (net n36 (joined (portRef OUT (instanceRef adder_6))
      (portRef B (instanceRef celement2_10))))
    (net n37 (joined (portRef OUT (instanceRef register_5))
      (portRef B (instanceRef adder_7))))
    (net n38 (joined (portRef OUT (instanceRef adder_1))
      (portRef A (instanceRef adder_7))))
    (net n39 (joined (portRef OUT (instanceRef adder_7))
      (portRef B (instanceRef adder_8))))
    (net n40 (joined (portRef OUT (instanceRef adder_1))
      (portRef A (instanceRef adder_8))))
    (net n41 (joined (portRef OUT (instanceRef adder_8))
      (portRef A (instanceRef celement2_11))))
    (net n42 (joined (portRef OUT (instanceRef adder_8))
      (portRef B (instanceRef celement2_11))))
    (net n43 (joined (portRef OUT (instanceRef register_6))
      (portRef A (instanceRef adder_9))))
    (net n44 (joined (portRef OUT (instanceRef adder_7))
      (portRef A (instanceRef adder_10))))
    (net n45 (joined (portRef OUT (instanceRef adder_9))
      (portRef B (instanceRef adder_10))))
    (net n46 (joined (portRef OUT (instanceRef celement2_7))
      (portRef A (instanceRef celement2_4))))
    (net n47 (joined (portRef OUT (instanceRef celement2_8))
      (portRef B (instanceRef celement2_4))))
    (net n48 (joined (portRef OUT (instanceRef celement2_5))
      (portRef A (instanceRef celement3_5))))
    (net n49 (joined (portRef OUT (instanceRef celement2_6))
      (portRef B (instanceRef celement3_5))))
    (net n50 (joined (portRef OUT (instanceRef celement2_4))
      (portRef C (instanceRef celement3_5))))
    (net n51 (joined (portRef OUT (instanceRef celement3_5))
      (portRef A (instanceRef celement2_6))))
    (net n52 (joined (portRef OUT (instanceRef adder_10))
      (portRef B (instanceRef celement2_6))))
    (net n53 (joined (portRef OUT (instanceRef celement2_6))
      (portRef B (instanceRef xor2_2))))
    (net n54 (joined (portRef OUT (instanceRef celement2_5))
      (portRef B (instanceRef xor2_1))))
    (net n55 (joined (portRef OUT (instanceRef celement2_9))
      (portRef A (instanceRef dmux3_1))))
    (net n56 (joined (portRef OUT (instanceRef celement2_10))
      (portRef B (instanceRef dmux3_1))))
    (net n57 (joined (portRef OUT (instanceRef celement2_11))
      (portRef C (instanceRef dmux3_1))))
    (net n58 (joined (portRef OUT1 (instanceRef dmux3_1))
      (portRef A (instanceRef multiplier_1))))
    (net n59 (joined (portRef OUT2 (instanceRef dmux3_1))
      (portRef B (instanceRef multiplier_1))))
    (net n60 (joined (portRef OUT (instanceRef multiplier_1))
      (portRef B (instanceRef toggle3_1))))
    (net n61 (joined (portRef OUT (instanceRef toggle3_1))
      (portRef A (instanceRef register_5))))
    (net n62 (joined (portRef OUT (instanceRef toggle3_1))
      (portRef A (instanceRef register_6))))
    (net n63 (joined (portRef OUT (instanceRef toggle3_1))
      (portRef A (instanceRef register_7))))
    (net n64 (joined (portRef OUT (instanceRef xor2_2))
      (portRef A (instanceRef celement2_12))
      (portRef A (instanceRef celement2_13))
      (portRef A (instanceRef celement2_14))
      (portRef A (instanceRef celement2_15))
```

```
    (portRef A (instanceRef celement2_16))
    (portRef A (instanceRef celement2_17))
    (portRef A (instanceRef celement2_18))))
(net n65 (joined (portRef OUT (instanceRef celement2_18))
    (portRef A (instanceRef register_12))))
(net n66 (joined (portRef OUT (instanceRef celement2_17))
    (portRef A (instanceRef register_11))))
(net n67 (joined (portRef OUT (instanceRef celement2_16))
    (portRef A (instanceRef register_10))))
(net n68 (joined (portRef OUT (instanceRef celement2_15))
    (portRef A (instanceRef register_9))))
(net n69 (joined (portRef OUT (instanceRef celement2_14))
    (portRef A (instanceRef register_8))))
(net n70 (joined (portRef OUT (instanceRef xor2_6))
    (portRef B (instanceRef celement2_12))))
(net n71 (joined (portRef OUT (instanceRef xor2_7))
    (portRef B (instanceRef celement2_13))))
(net n72 (joined (portRef OUT (instanceRef ))
    (portRef A (instanceRef celement3_2))))
(net n73 (joined (portRef OUT (instanceRef ))
    (portRef B (instanceRef celement3_2))))
(net n74 (joined (portRef OUT (instanceRef register_8))
    (portRef A (instanceRef celement2_7))))
(net n75 (joined (portRef OUT (instanceRef register_9))
    (portRef B (instanceRef celement2_7))))
(net n76 (joined (portRef OUT (instanceRef register_8))
    (portRef A (instanceRef adder_11))))
(net n77 (joined (portRef OUT (instanceRef register_9))
    (portRef B (instanceRef adder_11))))
(net n78 (joined (portRef OUT (instanceRef register_10))
    (portRef A (instanceRef adder_12))))
(net n79 (joined (portRef OUT (instanceRef adder_11))
    (portRef B (instanceRef adder_12))))
(net n80 (joined (portRef OUT (instanceRef register_11))
    (portRef A (instanceRef celement2_13))))
(net n81 (joined (portRef OUT (instanceRef register_12))
    (portRef B (instanceRef celement2_13))))
(net n82 (joined (portRef OUT (instanceRef register_11))
    (portRef A (instanceRef adder_13))))
(net n83 (joined (portRef OUT (instanceRef register_12))
    (portRef B (instanceRef adder_13))))
(net n84 (joined (portRef OUT (instanceRef adder_13))
    (portRef A (instanceRef adder_14))))
(net n85 (joined (portRef OUT (instanceRef xor2_8))
    (portRef B (instanceRef adder_14))))
(net n86 (joined (portRef OUT (instanceRef adder_14))
    (portRef A (instanceRef muluplier_2))))
(net n87 (joined (portRef OUT (instanceRef adder_14))
    (portRef B (instanceRef muluplier_2))))
(net n88 (joined (portRef OUT (instanceRef muluplier_2))
    (portRef A (instanceRef adder_15))))
(net n89 (joined (portRef OUT (instanceRef xor2_8))
    (portRef B (instanceRef adder_15))))
(net n90 (joined (portRef OUT (instanceRef adder_15))
    (portRef A (instanceRef adder_16))))
(net n91 (joined (portRef OUT (instanceRef adder_14))
    (portRef B (instanceRef adder_16))))
(net n92 (joined (portRef OUT (instanceRef adder_15))
    (portRef B (instanceRef celement2_12))))
(net n93 (joined (portRef OUT (instanceRef celement2_9))
    (portRef A (instanceRef celement2_12))))
(net n94 (joined (portRef OUT (instanceRef celement2_12))
    (portRef A (instanceRef xor2_8))))
(net n95 (joined (portRef OUT (instanceRef celement2_2))
    (portRef A (instanceRef celement2_9))))
(net n96 (joined (portRef OUT (instanceRef celement2_13))
    (portRef B (instanceRef celement2_9))))
(net n97 (joined (portRef OUT (instanceRef celement2_7))
    (portRef A (instanceRef celement2_8))))
(net n98 (joined (portRef OUT (instanceRef register_10))
    (portRef B (instanceRef celement2_8))))
(net n99 (joined (portRef OUT (instanceRef celement3_2))
```

```
(portRef A (instanceRef celement2_9))))
(net n100 (joined (portRef OUT (instanceRef celement2_13))
(portRef B (instanceRef celement2_8))))
(net n101 (joined (portRef OUT (instanceRef adder_12))
(portRef A (instanceRef celement2_10))))
(net n102 (joined (portRef OUT (instanceRef celement2_9))
(portRef B (instanceRef celement2_10))))
(net n103 (joined (portRef OUT (instanceRef celement2_10))
(portRef A (instanceRef xor2_7))))
(net n104 (joined (portRef OUT (instanceRef adder_16))
(portRef B (instanceRef celement2_11))))
(net n105 (joined (portRef OUT (instanceRef celement2_9))
(portRef A (instanceRef celement2_11))))
(net n106 (joined (portRef OUT (instanceRef celement2_11))
(portRef A (instanceRef xor2_6))))
(net n107 (joined (portRef OUT (instanceRef xor2_3))
(portRef A (instanceRef celement2_14))))
(net n108 (joined (portRef OUT (instanceRef xor2_3))
(portRef A (instanceRef celement2_15))))
(net n109 (joined (portRef OUT (instanceRef celement2_14))
(portRef A (instanceRef register_13))))
(net n110 (joined (portRef OUT (instanceRef celement2_15))
(portRef A (instanceRef register_14))))
(net n111 (joined (portRef OUT (instanceRef register_13))
(portRef A (instanceRef adder_17))))
(net n112 (joined (portRef OUT (instanceRef register_14))
(portRef A (instanceRef celement2_23))))
(net n113 (joined (portRef OUT (instanceRef register_14))
(portRef B (instanceRef celement_23))))
(net n114 (joined (portRef OUT (instanceRef adder_17))
(portRef A (instanceRef adder_19))))
(net n115 (joined (portRef OUT (instanceRef register_13))
(portRef A (instanceRef celement_23))))
(net n116 (joined (portRef OUT (instanceRef adder_17))
(portRef B (instanceRef celement_23))))
(net n117 (joined (portRef OUT (instanceRef ))
(portRef A (instanceRef celement_19))))
(net n118 (joined (portRef OUT (instanceRef register_15))
(portRef B (instanceRef celement_19))))
(net n119 (joined (portRef OUT (instanceRef celement2_19))
(portRef A (instanceRef celement2_17))))
(net n120 (joined (portRef OUT (instanceRef adder_18))
(portRef B (instanceRef celement_17))))
(net n121 (joined (portRef OUT (instanceRef adder_18))
(portRef B (instanceRef adder_20))))
(net n122 (joined (portRef OUT (instanceRef adder_19))
(portRef A (instanceRef celement_24))))
(net n123 (joined (portRef OUT (instanceRef adder_19))
(portRef B (instanceRef celement_24))))
(net n124 (joined (portRef OUT (instanceRef celement2_14))
(portRef A (instanceRef adder_21))))
(net n125 (joined (portRef OUT (instanceRef register_16))
(portRef B (instanceRef adder_21))))
(net n126 (joined (portRef OUT (instanceRef adder_21))
(portRef A (instanceRef celement2_18))))
(net n127 (joined (portRef OUT (instanceRef celement2_18))
(portRef A (instanceRef celement2_20))))
(net n128 (joined (portRef OUT (instanceRef celement2_17))
(portRef B (instanceRef adder_20))))
(net n129 (joined (portRef OUT (instanceRef celement2_20))
(portRef A (instanceRef celement2_22))))
(net n130 (joined (portRef OUT (instanceRef adder_25))
(portRef B (instanceRef celement2_22))))
(net n131 (joined (portRef OUT (instanceRef adder_21))
(portRef A (instanceRef adder_22))))
(net n132 (joined (portRef OUT (instanceRef adder_18))
(portRef B (instanceRef adder_21))))
(net n133 (joined (portRef OUT (instanceRef adder_22))
(portRef A (instanceRef adder_23))))
(net n134 (joined (portRef OUT (instanceRef xor2_9))
(portRef B (instanceRef adder_23))))
(net n135 (joined (portRef OUT (instanceRef adder_23))
```

```
    (portRef A (instanceRef celement2_25))))
(net n136 (joined (portRef OUT (instanceRef adder_23))
    (portRef B (instanceRef celement2_25))))
(net n137 (joined (portRef OUT (instanceRef celement2_25))
    (portRef A (instanceRef register_17))))
(net n138 (joined (portRef OUT (instanceRef register_17))
    (portRef A (instanceRef adder_25))))
(net n139 (joined (portRef OUT (instanceRef xor2_9))
    (portRef B (instanceRef adder_25))))
(net n140 (joined (portRef OUT (instanceRef adder_25))
    (portRef A (instanceRef adder_24))))
(net n141 (joined (portRef OUT (instanceRef register_23))
    (portRef B (instanceRef adder_25))))
(net n142 (joined (portRef OUT (instanceRef celement2_23))
    (portRef A (instanceRef dmux3_2))))
    .43 (joined (portRef OUT (instanceRef celement2_24))
    (portRef B (instanceRef dmux3_2))))
(net n144 (joined (portRef OUT (instanceRef celement2_25))
    (portRef C (instanceRef dmux3_2))))
(net n145 (joined (portRef OUT1 (instanceRef dmux3_2))
    (portRef A (instanceRef multiplier_3))))
(net n146 (joined (portRef OUT2 (instanceRef dmux3_2))
    (portRef B (instanceRef multiplier_3))))
(net n147 (joined (portRef OUT (instanceRef multiplier_3))
    (portRef B (instanceRef toggle3_2))))
(net n148 (joined (portRef OUT (instanceRef toggle3_2))
    (portRef A (instanceRef register_15))))
(net n149 (joined (portRef OUT (instanceRef toggle3_2))
    (portRef A (instanceRef register_16))))
(net n150 (joined (portRef OUT (instanceRef toggle3_2))
    (portRef A (instanceRef register_17))))
(net n151 (joined (portRef OUT (instanceRef xor2_4))
            (portRef A (instanceRef celement2_26))))
(net n152 (joined (portRef OUT (instanceRef xor2_16))
            (portRef A (instanceRef celement2_26))))
(net n153 (joined (portRef OUT (instanceRef xor2_4))
            (portRef A (instanceRef celement2_27))))
(net n154 (joined (portRef OUT (instanceRef celement2_27))
            (portRef A (instanceRef register_18))))
(net n155 (joined (portRef OUT (instanceRef register_18))
            (portRef A (instanceRef adder_26))))
(net n156 (joined (portRef OUT (instanceRef adder_26))
            (portRef A (instanceRef multiplier_4))))
(net n157 (joined (portRef OUT (instanceRef adder_26))
            (portRef B (instanceRef multiplier_4))))
(net n158 (joined (portRef OUT (instanceRef multiplier_4))
            (portRef A (instanceRef celement2_30))))
(net n159 (joined (portRef OUT (instanceRef celement2_30))
            (portRef A (instanceRef celement2_28))))
(net n160 (joined (portRef OUT (instanceRef multiplier_4))
            (portRef A (instanceRef adder_27))))
(net n156 (joined (portRef OUT (instanceRef xor2_16))
            (portRef B (instanceRef adder_27))))
```

Chapter 6

Conclusion

A subset of the occam language has been chosen as a specification language for high-level synthesis. To bridge the program level and circuit level of abstraction, an intermediate form (IFORM) has been used in the synthesis. A set of requirements has been [illegible] for such an IFORM. The IFORM of representation, a marked graph with modified [illegible]antics, has been presented as a semantic model for occam programs. The IFORM captures arbitration (alternation statement) and synchronous communication. By means of the IFORM, a formal definition is given to "a circuit implements an occam program". A time-sharing theorem with proof has been presented. Techniques to find time-sharable candidates has been provided with the use of parse tree and the IFORM. Augmentations of the IFORM by data flow extraction has been presented. Such augmentations have been found useful in removing some of the control sequence without violating the correctness criteria. Transitions which are neither sequential nor mutually exclusive cannot time-share a circuit component. Augmentation techniques to serialize concurrent transitions have been presented. Care must be taken in serializing transitions. Possible violations of incorrect augmentation have been provided. The intermediate form is also used for the evaluation of the performance of the circuit synthesized from it. Since the firing rules have been modified, previous approaches for measuring performance of concurrent systems cannot be extended to the IFORM. For a sub class of IFORMs a method based on the previous approach has been presented. Finally, a strategy for exploring the optimization search space is outlined. A fifth order elliptic filter is used as a demonstration of the proven results as well as the conjectured results. A synthesis software has been developed as an implementation of the synthesis methodology presented in this thesis. The software has been used in the synthesis of a fifth order wave elliptic filter. The EDIF netlist generated by the software can be used along with a basic cell library, to generate circuit layout for chip fabrication. CADENCE's CDS design system has been used in generating the layout for the elliptic filter example by the place and route utility.

As future extension of this thesis, more correctness preserving augmentation strategies can be explored. A method for measuring performance which takes care of modified firing rules in the IFORM has to be worked out. Translation and optimizations methods could be extended for circuits derived for other architectures from an occam program. Other specification languages can be explored.

Chapter 7

References

[1] C.G. Bell and A. Newell, *Computer Structures: Readings and Examples*. New York, NY: McGraw-Hill, 1971.

[2] C.H. van Berkel and R. Saeijs, "Compilation of Communicating Processes into Delay-Insensitive Circuits", *International Conference on Computer* Design, 1988, pp. 157-162.

[3] G.M. Brown, "Towards Truly Delay-Insensitive Circuit Realization of Process Algebras", manuscript.

[4] E. Brunvand and R.F. Sproull, "Translating Concurrent Programs into Delay-Insensitive Circuits", *International Conference on Computer-Aided-Design*, 1989, pp. 262-265.

[5] S.M. Burns and A.J. Martin, "Syntax-directed Translation of Concurrent Programs into Self-timed Circuits", *Advanced Research in VLSI*, 1988, pp. 35-50.

[6] S.M. Burns and A.J. Martin, "Performance Analysis and Optimization of Asynchronous Circuits", *Advanced Research in VLSI*, 1990, pp. 71-86.

[7] M.R. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and its Applications," IEEE Trans. on Computers, vol. C-30, no. 1, pp. 24-40, Jan. 1981.

[8] R. Camposano and W. Rosenstiel, "Synthesizing Circuits From Behavioural Descriptions," IEEE Trans. on CAD, vol. CAD-6, no 6, pp. 1098-1112, Nov. 1987.

[9] R. Camposano and R.M. Tablet, "Design Representation for the synthesis of Behavioral VHDL Models," in Proc. of the 9th In'l Conf. on CHDL. New York, NY: Elservier/North Holland, June 1989.

[10] K.M. Chandy and J. Misra, *Parallel Program Design: A Foundation*, Prentice Hall, 1989.

[11] J.S. Chiang and D. Radhakrishnan, "Hazard-free Design of Mixed Operating Mode Asynchronous Sequential Circuits", *International Journal of Electronics*, Vol. 68, No. 1, 1990, pp. 23-37.

[12] T.-A. Chu, "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications", *International Conference on Computer Design*, 1987, pp. 220-223.

[13] P. Cousot and R. Cousot, "Semantic Analysis of Communicating Sequential Processes", *Lecture Notes in Computer Science 85*, 1980, pp. 119-133.

[14] P. Dewilde, E. Deprettere, and R. Nouta, "Parallel and Pipelined VLSI Implementation of Signal Processing Algorithms", in Chapter 15, S.Y. Kung, H.J. Whitehouse, and T. Kailath (eds.), *VLSI and Modern Signal Processing*, Prentice Hall, 1985.

[15] J.C. Ebergen, *Translation of Programs into Delay-Insensitive Circuits*, Ph. D. Thesis, Eindhoven University of Technology, 1987.

[16] E.F. Girczcy, "Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions," *PhD Thesis, Carleton University, Ottawa, Canada*, July 1984.

[17] INMOS Limited, *Occam Programming Manual*, Prentice Hall, 1983.

[18] D.J. Kuch, R.H. Kuhn, B. Leasure, et. al, "The Structure of an Advanced Retargetable Vectorizer", *COMPSAC 80*, 1980.

[19] P.N. Lam and H.F. Li, "Hierarchical Design of Delay-Insensitive Systems", *IEE Proceedings, Part E: Computers and Digital Techniques*, Vol. 137, No. 1, Jan. 1990, pp. 41-56.

[20] L. Lavagno, K. Keutzer, and A.L. Sangiovanni-Vincentelli, "Synthesis of Verifiably Hazard-Free Asynchronous Control Circuits", *Advanced Research in VLSI*, 1991, pp. 87-102.

[21] H.F. Li, S.C. Leung, and P.N. Lam, "Synthesis of Delay-Insensitive Circuits by Refinement into Atomic Threads", *International Conference on Computer Design*, 1991, pp. 180-186.

[22] J. Magott, "Performance Evaluation of Concurrent Systems using Petri Nets", *Information Processing Letters*, Vol. 18, 1984, pp. 7-13.

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

[24] T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic Synthesis of Asynchronous Circuits from High-Level Specifications", *IEEE Transactions on Computer-Aided-Design*, Vol. 8, No. 11, 1989, pp. 1185-1205.

[25] G. De Micheli and D. C. Ku, "HERCULES: A System for High-Level Synthesis," in *Proc. of the 25th Design Automation Co*[illegible]. New York, NY: ACM/IEEE, June 1988, pp. 483-488.

[26] S. Nowick and D.L. Dill, "Synthesis of Asynchronous State Machines Using a Local Clock", *International Conference on Computer Design*, 1991, pp. 192-196.

[27] W. Peng and S. Purushothaman, "Towards Data Flow Analysis of Communicating Finite State Machines", *Symposium on Principles of Distributed Computing*, 1989, pp. 45-58.

[28] P.G. Paulin and J.P. Knight, "Force-Directed Scheduling in Automatic data path synthesis", *Proc. of the 24th Design Automation Conf.* New York, NY: ACM/IEEE, 1987, pp. 195-202.

[29] D.K. Probst and H.F. Li, "Modeling Reactive Hardware Processes using Partial Orders", *Semantics for Concurrency, Workshops in Computing*, July 1990, pp. 324-343.

[30] C.V. Ramamoorthy and G.S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems using Petri Nets", *IEEE Transactions on Software Engineering*, Vol. 6, No. 5, 1980, pp. 440-449.

[31] J.H. Reif and S.A. Smolka, "Data Flow Analysis of Distributed Communicating Processes", *International Journal of Parallel Programming*, Vol. 19, No. 1, 1990, pp. 1-30.

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

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

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

[35] J. Staunstrup and M.R. Greenstreet, "Designing Delay-Insensitive Circuits using Synchronized Transitions", *IMEC IFIP International Workshop on Applied Formal Methods for Correct VLSI Design*, 1989, pp. 741 - 758.

[36] I.E. Sutherland, "Micropipelines", *Communications ACM*, Vol. 32, No. 6, June 1989, pp. 720-738.

[37] H. Tricker, "Flamel: A High-Level Hardware Compiler," IEEE Tran. on CAD, vol. CAD-6, no. 2, pp. 259-269, Mar. 1987.

[38] 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.

[39] S.H. Unger, *Asynchronous Sequential Switching Circuits*, Wiley-Interscience, 1969.

[40] P. Vanbekbergen, F. Catthoor, G. Goossens, H. De Man, "Optimized Synthesis of Asynchronous Control Circuits from Graph-theoretic Specifications", *International Conference on Computer-Aided-Design*, 1990, pp. 184-187.

[41] K.J. Venkatesh, S.C. Leung, H.F. Li, and R. Jayakumar, "Optimizations in the Translation of Occam Programs into Delay-Insensitive Circuits", *Canadian Conference on VLSI*, 1990, 4A.4.1-4A.4.8.

[42] O. Yenersoy, "Synthesis of Asynchronous Machines Using Mixed Operation Mode", *IEEE Transactions on Computers*, Vol. 28, No. 4, 1979, pp. 325-329.

[43] G. Zimmermann, "MDS-The Mimola Design Method," J. Digital Sytems, vol. 4, no. 3, pp. 337-369, 1980.

Appendix A

Occam Subset: Syntax and Semantics

The source language used in this thesis is a subset of occam [5], a language based on CSP [8]. Occam programs are built from processes. The simplest process in an occam program is an action. An action is either an assignment, an input or an output process. The STOP process is not included in the occam subset as the program is being translated into a non terminating circuit implementation. Since the target architecture is delay-insensitive, WAIT process is not included in the occam subset. Repetitive constructs, functions and procedures are also not included in the occam subset.

7.1. Data types:

The data types have to be declared for all the variables used in the program using the VAR construct. The types supported are BIT, BYTE, INTEGER16, INTEGER32. Only one variable declaration is allowed in a program. VAR declaration has to be included if there is at least one variable to be declared. The subscript index specifies this requirement.

```
Syntax of variable declaration:
variable_declaration = VAR {1, <variables_type_list>}
<variables_type_list> = <variable_name_list> : <data_type>
<variable_name_list> = {1, <variable>}
<data_type> = <BIT> | <BYTE> | <INTEGER16> | <INTEGER32>
```

7.2. Channel declaration

All the channels used in the occam program have to be declared using the CHAN declaration statement. Along with the channel names, the types of those channels also have to be specified. Only one channel declaration is allowed in a program.

```
Syntax of channel declaration:
channel_declaration = VAR {1, <channels_type_list>}
<channels_type_list>    =    <channels_name_list>    :
<data_type>
<channels_name_list> = {1, <channel>}
```

7.3. Assignment

An assignment process assigns the value of an expression to a variable. In the occam subset we do not consider array variables and multiple assignment. Expressions may contain one or two operands operated by one unary or one binary operator. Consider the following example:

$$x := y + x$$

The expression y+x is evaluated and its value is assigned to the variable x.

```
Syntax of assignment process:
<variable_name> = <unary_operator> <variable_name> |
<variable_name>   =   <variable_name>   <binary_operator>
<variable_name>
<unary_operator> = - | NOT
<binary_operator> = +, -, *, /, <<, >>, AND, OR, XOR, <,
>, =, <=, >=, <>
```

7.4. Communication

Communication is an essential part of occam programs. Values are passed between concurrent processes by communication on channels. Each channel provides unbuffered unidirectional point-to-point communication between two concurrent processes. Communication is performed by an input action at the receiving process and an output process at the sending process. Communication without value passing can also be done by input and output actions by substituting the variable name with the keyword ANY.

7.4.1. Input action

An input action receives a value from a channel and assigns the received value to a variable. Consider the following example:

keyboard ? char

This process receives a value from the channel named keyboard and assigns the value to the variable char. The input action waits until a value is received.

```
Syntax of input action with value passing:
input = channel ? variable
Syntax of input action without value passing:
input = channel ? ANY
```

7.4.2. Output action

An output action transmits the value of a variable to a channel. In the occam subset only a variable can be used in the output process. Consider the following example:

screen ! char

This process transmits the value of the variable char to the channel named screen. The output action waits until the value has been received by a corresponding input.

```
Syntax of an output with value passing is:
output = channel ! variable
Syntax of an output without value passing is:
output = channel ! ANY
```

7.5. SKIP

The primitive process SKIP starts, performs no action and terminates.

```
Syntax of skip process is:
skip = SKIP
```

7.6. Control Constructs

Processes are composed from the primitive processes or some composed processes

using the following constructs.

7.6.1. Sequence

A sequence construct serializes the execution of the composing processes in the order in which they are specified. Consider the following example:

```
SEQ
  keyboard ? char
  screen ! char
```

This process combines two actions which are performed sequentially. The input keyboard ? char receives a value which is assigned to the variable char, then the following .: screen ! char is performed.

```
Syntax of sequence construct:
Sequence = SEQ
             process1
             process2
             ...
             processn
```

The keyword SEQ is followed by zero or more processes at an indentation of two spaces.

7.6.2. Conditional

A conditional combines two processes. One of which is guarded by a boolean variable and another guarded by TRUE boolean constant. When the boolean variable holds a value TRUE the process guarded by it is performed. Otherwise the process guarded by TRUE is performed.

```
Syntax of conditional process:
IF
  boolean variable
    process
  TRUE
    process
```

using the following constructs.

7.6.1. Sequence

A sequence construct serializes the execution of the composing processes in the order in which they are specified. Consider the following example:

```
SEQ
  keyboard ? char
  screen ! char
```

This process combines two actions which are performed sequentially. The input keyboard ? char receives a value which is assigned to the variable char, then the following .: screen ! char is performed.

Syntax of sequence construct:

$$\text{Sequence} = \text{SEQ} \quad \begin{array}{l} \text{process}_1 \\ \text{process}_2 \\ \ldots \\ \text{process}_n \end{array}$$

The keyword SEQ is followed by zero or more processes at an indentation of two spaces.

7.6.2. Conditional

A conditional combines two processes. One of which is guarded by a boolean variable and another guarded by TRUE boolean constant. When the boolean variable holds a value TRUE the process guarded by it is performed. Otherwise the process guarded by TRUE is performed.

Syntax of conditional process:

```
IF
 boolean variable
   process
 TRUE
   process
```

Processes are indented by two spaces.

7.6.3. Loop

A loop repeats a process while an associated boolean variable is true. Consider the following example:

```
WHILE not_eof
  SEQ
    in ? buffer
    out ! buffer
```

This loop repeatedly copies a value from the channel in to the channel out. The copying continues while the boolean not_eof is true. The sequence is not performed if the boolean is initially false. Since the process behavior will be implemented by circuit, the n program will be expected to contain at least one WHILE construct with boolean constant TRUE. For the purpose of performance estimation, no nested WHILE loops are allowed in the occam subset and the boolean variable has to be the boolean constant TRUE.

```
Syntax
loop = WHILE boolean
           process
```

7.6.4. Parallel

The parallel combines a number of processes which are performed concurrently. Consider the following example:

```
WHILE TRUE
  SEQ
    x := next
    PAR
      in ? next
      SEQ
        x := x * x
        out ! x
```

The parallel inputs the next value to be processed from one channel while the last

value is being processed and output on another.

```
Syntax
parallel = PAR
                process1
                .
                processn
```

The keyword PAR is followed by one or more processes at an indentation of two spaces.

7.6.5. Alternation

An alternation combines two processes. One process is guarded by a boolean and an input. Another process is guarded by the SKIP primitive. The alternation performs the process associated with the guard which is ready (SKIP is always ready) and the boolean which is true. If both guard input is ready and boolean is true, then only the input/SKIP (if that is the guard) and the associated process is performed. Consider the following example:

```
ALT
  bool & in ? data
    out ! data
  SKIP
    out ! no_data
```

SKIP is treated as though it where a ready input, and may be selected immediately. If the input in ? data is also ready, only one of the processes is performed, which process will be performed is undefined.

```
Syntax of alternation construct:
ALT
  boolean & input
    process
  SKIP
    process
```

7.7. Program

An occam subset program is given a name using the keyword PROGRAM. After the program name declaration, the VAR and CHAN declarations are specified one after the other without any indentations. After the declarations the occam process is specified.

Syntax of an occam program:

```
program = PROGRAM program_name
              variable declaration
              channel declaration
                 process
```

Example Occam program:

```
PROC ELLIPTICFILTER(in, out)
    VAR x1,x2,x3,x4,x5,x6,x7,x8,x9
    CHAN in, out, a, b, c
    PAR
        WHILE TRUE
            SEQ
                in? x1
                x2 : = x1 * x1
                out ! x2
        WHILE TRUE
            ALT
                out ? x3
                    IF x3 > 0
                        a ! x3
                    TRUE
                        a ! 0
                b ? x4
                    in ! x4
```

Appendix B

Basic circuit elements

A.1. Fork

A forked wire performs corresponding to the fork F transition. The graphical representation of forked wire is:

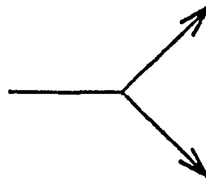


Fig. B.1. Fork

A.2. C-element

The C-element (also called as Muller C-element) performs like an AND element for transition signaling. When both inputs of a C-element are in the same logical state, the C-element's state and its output are copies of that state. When the two inputs differ, the C-element uses internal storage to retain its previous state and holds its output unchanged. Thus only after an event takes place on both of its inputs will a C-element produce an event at its output. The C-element generalizes easily to three or more inputs, requiring that all of them reach a new logical state before copying that state as output. The graphical representation of a C-element is:

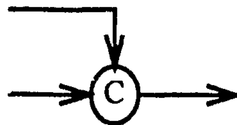


Fig. B.2. C-element

A.3. Exclusive or (XOR)

The exclusive or (XOR) circuit acts as the OR element for events. When either input of an XOR circuit changes state, its output also changes state. Thus an event received on either the first OR the second input of the XOR will produce an output event. The graphical representation of an XOR is:

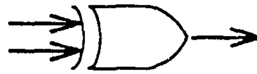


Fig. B.3. XOR

A.4. Toggle

The toggle circuit produces events alternately on its two outputs, starting with the dot, in response to events at its input. The graphical representation of a toggle element is:

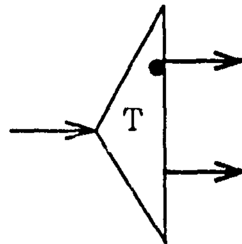


Fig. B.4. Toggle

A.5. Select

The select element steers an incoming event to one output or the other depending on the value of a data input. The boolean value must be available before the incoming event that it steers. The graphical representation of a select element is:

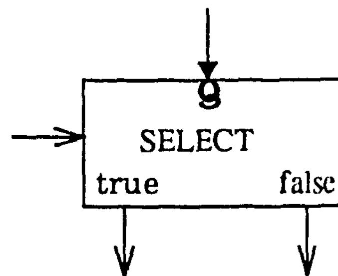


Fig. B.5. Select

A.6. Register

A register is used in the translation of data input process. It is also used while time-sharing data manipulating operators. A signal transition at its input latches the data value bundled with that wire and initiates a transition at its output. The graphical representation of a register is:

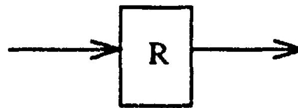


Fig. B.6. Register

A.7. CALL

The call element remembers which of its inputs most recently received an event, and returns an event to the matching output terminal after a called procedure has finished. The call element operates properly only if each call completes before a subsequent call occurs. The graphical representation of a call module is:

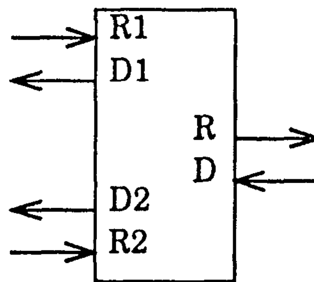


Fig. B.7. Call

A.8. BCALL

BCALL is a variant of the call module. It provides the called procedure with the opportunity to return a boolean value with its acknowledgment by choosing among two

acknowledgment signals. The graphical representation of a BCALL module is:

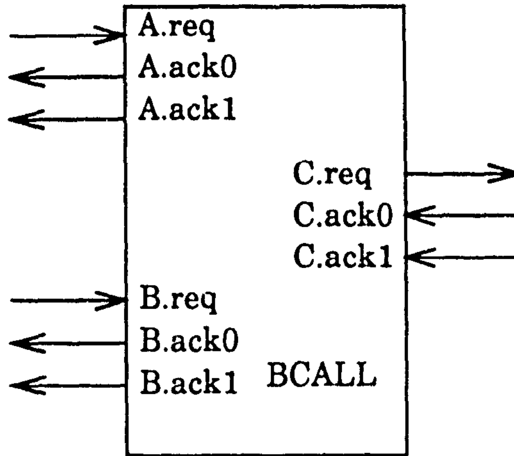


Fig. B.8. Bcall

A.9. Arbiter

The arbiter decides between two events whose arrival sequence is unknown, pro-ng a grant event for only one of them even if they arrive at very nearly the same time. Like a semaphore in programming, it delays subsequent grants until after receiving an event on the done wire (D) corresponding to an earlier grant so that only one grant at a time is ever outstanding. The graphical representation of an Arbiter module is:

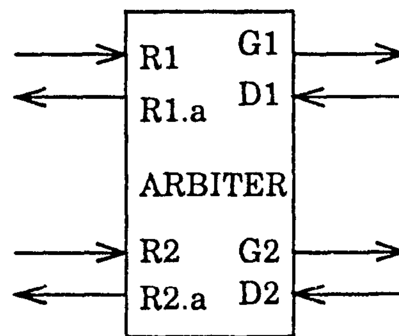
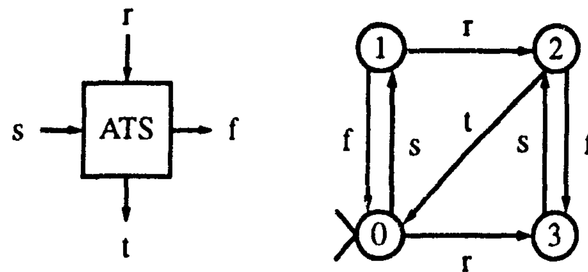


Fig. B.9. Arbiter

A.10. Arbitrating Test and Set (ATS)

The state of an Arbitrating test and set (ATS) module is initially 1 and is reset to 0

by a transition on its R input. The current state is tested and set to 1 by a transition on its T input. An ATS can be constructed using arbiters and de-multiplexers. The graphical representation and finite state machine specification of an ATS module is:

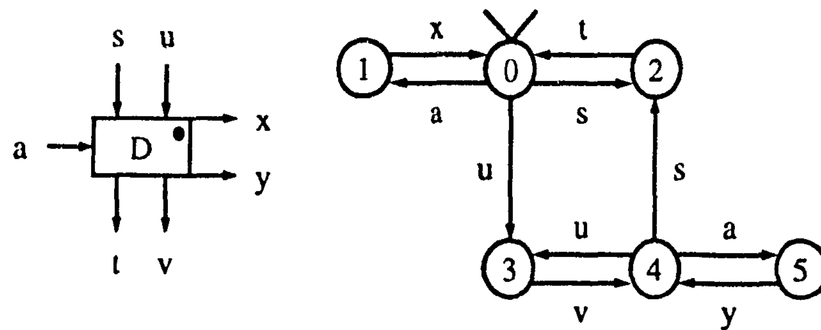


Finite state machine specification

Fig. B.10. ATS

A.11. De-multiplexer

A de-multiplexer is used as a storage unit for events. The graphical representation and finite state machine specification of a de-multiplexer module is:



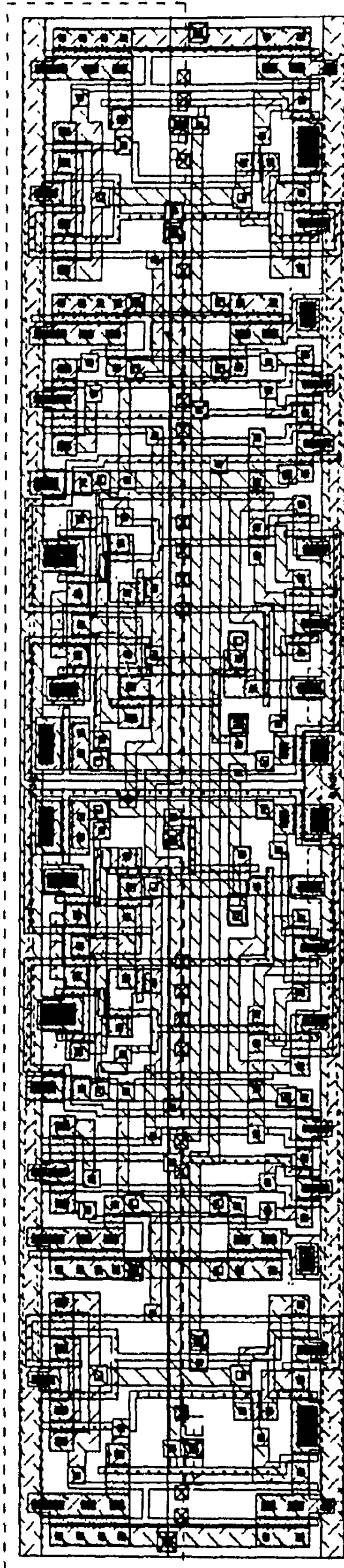
A de-multiplexer and its finite state machine specification

Fig. B.11. De-multiplexer

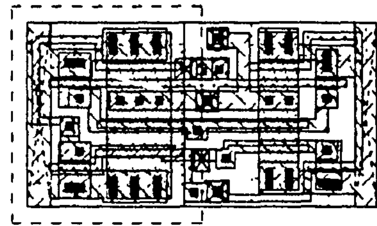
Appendix C

Basic DI Cell Library

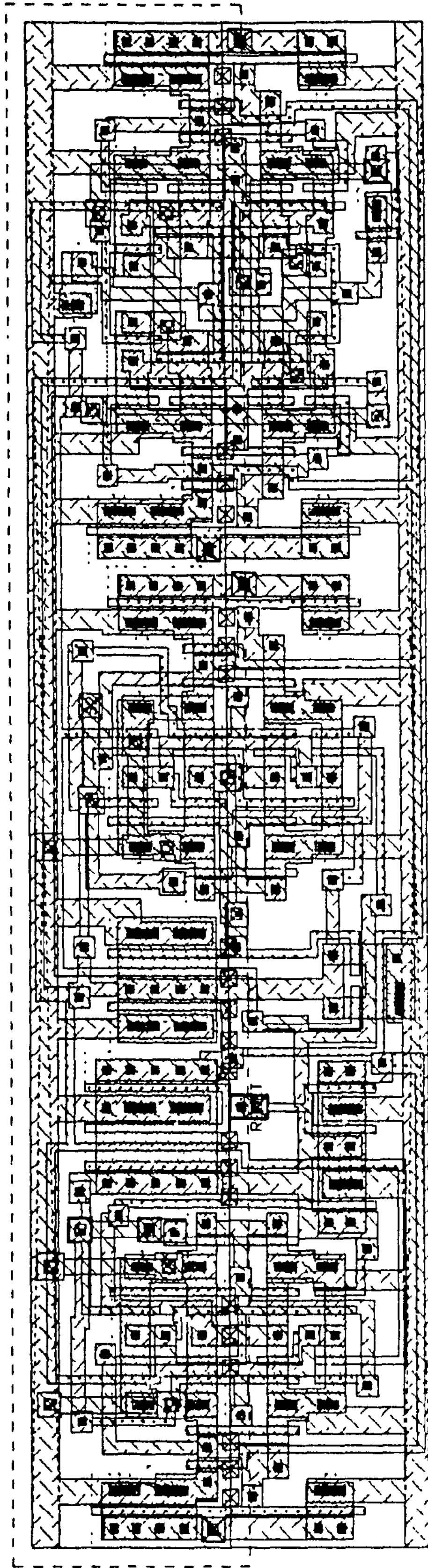
The basic DI cells were designed by the VLSI research group in the computer science department of Concordia University. The cells were designed under CADENCE CAD system environment using NT's CMOS4S 1.2 micron technology. In building the layout for elliptic filter, apart from the DI cells, basic cells from CMC have also been used. The circuit layouts for the DI cells are given in pages 76-77

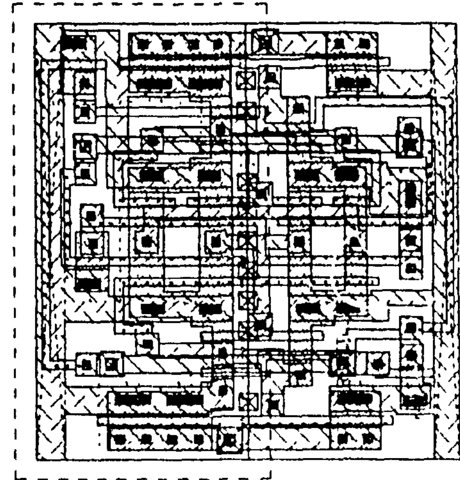


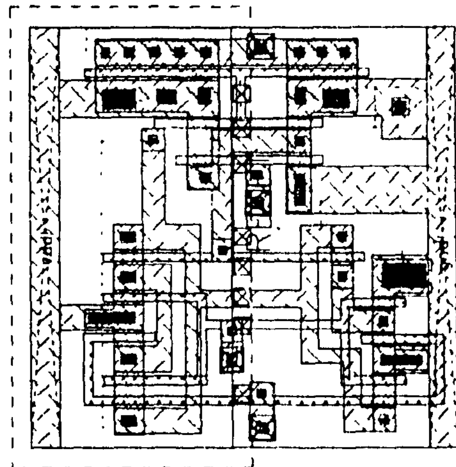
arbiter



xor2


DEMUX


TOGGLE


celement