AN ABSTRACT OF THE THESIS OF

Lalit T. Merani for the degree of Master of Science in Electrical and Computer Engineering presented on August 24, 1993.

Title: MICRO DATA FLOW (MDF): A Data Flow Approach to Self-timed VLSI System Design for DSP

Abstract approved: Redacted for Privacy

Shih-Lien Lu

Synchronization is one of the important issues in digital system design. While other approaches have been intriguing, up until now a globally clocked timing discipline has been the dominant design philosophy. However, we have reached the point, with advances in technology, where other options should be given serious consideration. VLSI promises great processing power at low cost. This increase in computation power has been obtained by scaling the digital IC process. But as this scaling continues, it is doubtful that the advantages of faster devices can be fully exploited. This is because the clock periods are getting much smaller in relation to the interconnect propagation delays, even within a single chip and certainly at the board and backplane level.

In this thesis, some alternative approaches to synchronization in digital system design are described and developed. We owe these techniques to a long history of effort in both digital computational system design as well as digital communication system design. The latter field is relevant because large propagation delays have always been a dominant consideration in its design methods.

Asynchronous design gives better performance than comparable synchronous design in situations for which a global synchronization with a high speed clock becomes a constraint for greater system throughput. Asynchronous circuits with unbounded gate delays, or self-timed digital circuit can be designed by employing either of two request-acknowledge protocols - 4-cycle and 2-cycle.

We will also present an alternative approach to the problem of mapping computation algorithms directly into asynchronous circuits. Data flow graph or language is used to describe the computation algorithms. The data flow primitives have been designed using both the 2-cycle and 4-cycle signaling schemes which are compared in terms of performance and transistor count. The 2-cycle implementations prove to be better than their 4-cycle counterparts.

A promising application of self-timed design is in high performance DSP systems. Since there is no global constraint of clock distribution, localized forward-only connection allows computation to be extended and sped up using pipelining. A decimation filter was designed and simulated to check the system level performance of the two protocols. Simulations were carried out using VHDL for high level definition of the design. The simulation results will demonstrate not only the efficacy of our synthesis procedure but also the improved efficiency of the 2-cycle scheme over the 4-cycle scheme.

MICRO DATA FLOW (MDF):
A Data Flow Approach to Self-timed VLSI System Design for DSP

by

Lalit T. Merani

A THESIS

submitted to

Oregon State University

in partial fulfillment of

the requirements for the degree of

Master of Science

Completed August 24, 1993

Commencement June, 1994

APPROVED:

Redacted for Privacy

Professor of Electrical and Computer Engineering in charge of major

Redacted for Privacy

Head of the Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented August 24, 1993

Typed by researcher for Lalit T. Merani

ACKNOWLEDGEMENTS

This report describes research done at the Department of Electrical and Computer Engineering at Oregon State University. Support for the research was provided by the National Science Foundation grant # MIP-9211510. This research has also been helped by *View*logic Inc.

I would like to thank my major professor Dr. Shih-Lien Lu. His contribution to this thesis is irreplaceable (most of the ideas were his) but represents only a small fraction of how he has helped me in the last two years. Shih-Lien's clear and simple expression of his thorough understanding of many fields and his enthusiasm for knowledge makes him one of the best teachers I have ever had.

I would like to acknowledge Prof. Bella Bose's help and inspiration. He has, in his quiet way, influenced my attitude and enthusiasm. He has taught me, in the words of M. A. Biot, "the tradition of clarity, simplicity, intuitive understanding, unpretentious depth, and a shunning of the irrelevant."

I thank Dr. Griffiths and Dr. Shor for having taken time out of their busy schedules to be available for my thesis defense. I would like to thank the following people for invaluable discussions and various other help: Pat Lenders, Jim Herzog, Sayfe Kiaei, Ben Lee, Rita Wells, Tarek Abdelrahman, Suzy Rogers, Chih-Ming Chang. A special mention of thanks to Ravi Ramachandran for his invaluable help in drawing the circuit schematics and through the various draft stages of this report.

Further, I would like to thank my personal friends, Meenu, Shree, Satish, Ashok, Sailesh, Vivek, Manoj, Shashank, Varad, Sunil, Qazi for all the good times that made this stay at OSU easier.

Finally, to my parents, thanks for your love and blessings, without which this would simply have not been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

MICRO DATA FLOW (MDF):

A Data Flow Approach to Self-timed VLSI System Design for DSP

Chapter 1. SYNCHRONIZATION

1.0 Introduction

The last two decades and in essence the '80s have observed the realization of the visions of science fiction writers, considered fanciful at best, even twenty years ago. One of those visions involved the creation of an information society. This required the storage, transformation and communication of information. The main force behind these great technological advances has been the digital model for system design. System here includes both computer systems and communication systems.

In the world of computers and computing, there is always an imbalance between the supply and demand of computing power. Problems which occur in the fields of meteorology, image processing, global models, wind tunnel simulation and simulation of computer systems, among others, are examples of a class of problems which demand very high computing power. Even with the impressive choice of machines that is available to us today, we find ourselves at a great disadvantage when trying to solve such problems.

In recent years, we have seen a veritable explosion of VLSI-based solutions for digital signal and image processing. With the advent of multi-media, the computation and speed requirements of application specific digital systems, which up until now slowly increased, will now steeply rise. Digital cellular telephony, High Definition Television, re-recordable optical memories will all contribute to this rise.

Thus, in both the worlds of general-purpose and application specific computing the speed requirements are reaching a stage at which it would be worth evaluating the basic assumptions of current digital system design models.

We are also interested in systems that scale up as the size of the problem increases. This flexibility is going to be very important in the design of future digital systems. We need to look at models that lend themselves very naturally to these requirements of scaling. Linked to these reuiremnets are issues of power consumption, as we become increasingly aware of our environmental responsibilities.

The problem of developing new models is definitely non-trivial. But, where do we begin our re-learning? A basic assumption is that do any effective computing we require a global clock. Like the communications industry, can the computing industry divest itself of this self-imposed restriction of a global clock? In the next section, we discuss the basic issue of synchronization and the effect it has on the way we have designed digital computing systems up until now.

1.1 Asynchronous and Synchronous Digital System Design

Synchronization is one of the important issues in digital system design, especially in the effective design of any large computer and communication system. While other approaches have been intriguing, up until now a globally clocked timing discipline has been the dominant design philosophy. Some of the important factors that influenced the methodology of digital system design to take this course were:

- Major components of a computer system, like the memory and the ALU, were built synchronously.

- With global synchronization, the circuit transients do not affect the proper operation of the whole system.
- The step - by - step nature of synchronous systems made it easy to design and trace the sequence of actions performed by them.
- Synchronous systems were favored because they required fewer gates, which meant a lower cost if the systems were built from gates.

In fact, the reasons that contributed to the control flow, or von Neumann, model of digital computer system design becoming the widely accepted norm in the design of digital computers also contributed to the computer architects at that time favoring global synchronization. But digital communication continued to employ asynchronous design philosophy. The main reason for this was that the clock period was many orders of magnitude smaller than the communication delays.

However, we have reached a point, with advances in VLSI technology, where global synchronization does not suffice. This is because the clock periods are getting much smaller than the interconnect propagation delays within a single chip, not to mention at the board and backplane level. At the same time the theory and methodology of asynchronous computing system design has been maturing. So an intuitive first look suggests that asynchronous computing system design is worth a closer scrutiny.

Besides this, asynchronous computing system design offers other advantages [1]. First of all, asynchronous designs are algorithmic. It is easier to convert an algorithm to a wiring list for asynchronous modules than translating the algorithm into step by step procedures. Secondly, speed independent modules allow the system to perform correctly. There is no need to adjust the pulse width and clock period to fit all modules' timing requirements. It will avoid the clock skewing problem. Third,

the speed of execution is taken to be as fast as the problem or algorithm will allow. Fourth, composition of asynchronous modules in to asynchronous systems is readily simple. Building systems hierarchically is inherent. Building each individual asynchronous module on a single chip enables the testing and verification of each chip to be performed independently. With each module or chip verified to be functionally correct, they can be assembled on a single chip if area permits with no extra timing constraint needing to be satisfied. This ability to verify modules independently is becoming more and more desirable since system testing consumes larger and larger portions of the development cycle with each passing day. Fifth, incremental performance gains are easier to come by. Since there is no global timing requirement, any elements or blocks residing in the critical path may be replaced with faster counterparts without having to readjust the system timing parameters. Sixth, scaling up the system as the problem grows in size is much easier to accomplish. Finally, lower system noise and zero stand-by power consumption add to the many advantages listed before.

To contrast between the synchronous and asynchronous timing disciplines, we present an example which will illustrate the difference [2,3,4]. Pictorially, a synchronous system works like a scheduled train line. At every designated interval, there will be a train taking off from the station whether there is a full load of passengers or none at all. A particular passenger has to synchronize his/her travel itinerary with the schedule of the train. In contrast, an asynchronous system is like traveling in your own car. There is no fear of missing a scheduled departure time. There is no waiting in a depot for a train to arrive. You may visit a new place whenever you have finished visiting an old location.

This potential of asynchronous digital system design has many researchers in academia and some in industry interested. Researchers have a great liking for coining

new terms (as have we, by coining 'micro data flow'). As a result, asynchronous system design has also been called *delay-insensitive* system design. Here the delays of the interconnects within a chip and on the board were not taken into account. Other terms that can be seen in recent literature are *speed-independent* and *self-timed* digital system design. These both essentially mean the same. Both take into account the delays of the interconnects. This is very important as interconnect delays have become a very crucial issue in digital system design [5,6].

As can be understood from the simple analogy above, we are interested in the sequence of events. We no more allow the different computation blocks to abdicate their synchronization responsibility to a global clock. This requires the development of a protocol strategy, with the help of which we can accomplish synchronization. This protocol governs the proper relationship between events. In digital systems, this is implemented with signaling. We require a strategy based on some form of a request-acknowledge handshaking mechanism. Fig. 1.1 illustrates a block diagram of a request-acknowledge handshake scheme.

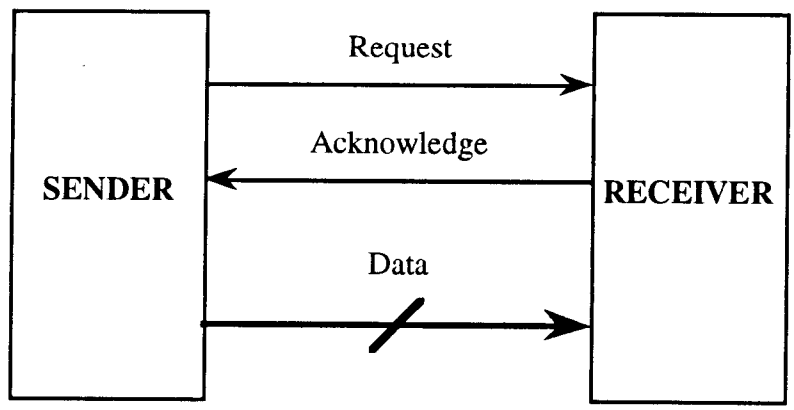

Figure 1.1 A block diagram viewpoint of a request-acknowledge handshake scheme.

Though, it is not apparent from the block diagram, it is very important to enforce a simple condition on this handshake scheme. The request needs to be bundled with the data, so that it does not arrive before valid data is ready for the receiver.

1.2 Organization of this document

In the next chapter we discuss the two different protocol strategies that are the most commonly used. We will also discuss the effect that pipelining has on digital system throughput. Chapter 3 consists of a discussion of data flow graphs adapted for signal processing functions. It will explain why a data flow graph is a natural method for high-level specification of computing algorithms that need to be mapped onto self-timed digital circuits. Chapter 4 is a brief discussion of how a hardware description language can be effectively used for the synthesis and analysis of self-timed circuits.

Chapter 5 concerns itself with the bulk of the self-timed digital circuits and contains most of the original contribution made by this thesis. In Chapter 6, we will use a decimation filter as an example to illustrate the performance issues involved. We will then conclude and hint at future directions of research in Chapter 7.

Chapter 2. PROTOCOLS AND PIPELINES

2.0 Introduction

To enable effective synchronization and also to make the computation blocks responsible for synchronization, we require a request-acknowledge protocol. We also know that in digital systems such a form of protocol can be implemented using signaling.

Seitz [3] illustrates two main request-acknowledge protocol strategies - a 4-cycle protocol (also known as Return-to-Zero protocol, or Muller signaling) and a 2-cycle protocol (also known as Non-Return-to-Zero protocol, or transition-signaling). In the case of the former protocol, we accomplish signaling by using pulses of indeterminate lengths. In the second protocol, we use transitions to achieve the same synchronization. In the rest of this thesis, these two strategies have been compared. This thesis should enable us to decide which of these protocols offers greater potential. But before we get to that stage we need to develop an understanding of the protocols and the related issues.

2.1 Pipelining

The pipeline is a common paradigm for high-speed computation. The analogy of a pipeline to the assembly line in an automobile factory is apt. The higher speed, or the greater throughput, is due to the fact that the different stages can act concurrently (i.e., after the pipeline has been filled, its latency has been overcome).

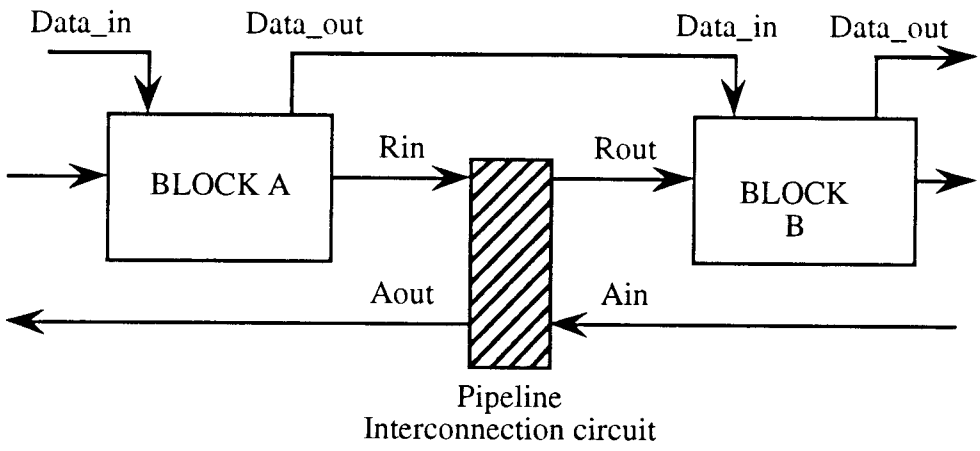

Figure 2.1 A simple example of a pipeline interconnection circuit that controls data transfer between blocks A and B.

Pipelines both store and process data, the storage elements and processing logic alternating along their length. Stripped of all processing logic, any pipeline acts like a series of storage elements through which data can pass. If the parts of the pipeline act in response to a widely distributed, global clock, then the pipeline is said to be clocked. If, on the other hand, the parts of the pipeline act independently as and when local events permit, the pipeline is said to be event-driven. Stripped of any processing logic, a pipeline acts like a First-In First-Out (FIFO) queue. We are interested in an event-driven pipeline with or without internal processing.

2.2 The Basic Model

Before we discuss the specifics of the different protocols, we will develop a basic model for a request-acknowledge protocol [7]. The simplest interconnection circuit, a pipelining handshake circuit shown in Fig. 2.1, checks the input request signal R_{in} (the completion signal of computation block A) to see if the output datum

of block A is valid, and checks the feedback acknowledge A_{in} to see if block B is ready for a new input. This is important since block A might take longer to finish than block B, which might compute data more than once. This is known as "continual feeding". If the situation reverses, data at the input of block B will be overwritten. Such is a "run-away" condition. R_{out} controls the request signal to block B, indicating when block B should start evaluation. A_{out} controls the acknowledge signal to the interconnection block preceding block A, notifying block A when its output datum is transferred to block B.

2.3 The 4-cycle Request-acknowledge Protocol

The common four-phase handshake protocol works as follows. Assume that the four signals R_{in}, R_{out} , A_{in}, and A_{out} are initially at logic level 0 (R_{in}^-, R_{out}^-, A_{in}^-, A_{out}^-). When block A finishes its computation, it raises $R_{in}(R_{in}^+)$ to request for a data transfer to block B. Since A_{in}, is initially low, meaning that block B is ready to accept a new input, the handshake circuit raises $A_{out}(A_{out}^+)$ to tell block A that its output datum has been accepted. R_{in} can then be reset (R_{in}^-). The handshake circuit then raises $R_{out}(R_{out}^+)$ to initiate the computation in block B. Eventually block B will complete its task and output a completion signal. This information is fed back through $A_{in}(A_{in}^+)$ so that $R_{out}(R_{out}^-)$ can be reset which will in turn reset $A_{in}(A_{in}^-)$ and complete the four-phase handshake loop. **The four-phase handshake protocol always uses the rising transitions to initiate operation and the falling transitions to reset.** The four-phase handshake protocol dictates that the sequence of signal transitions on the right hand side of the handshake circuit in Fig. 2.1 is always the iterative R_{out}^+ --> A_{in}^+ --> R_{out}^- --> A_{in}^- and on the left hand side R_{in}^+ --> A_{out}^+ --> R_{in}^- --> A_{out}^- .

2.4 The Transition Signaling Conceptual Framework

Ivan Sutherland[8] proposed a different timing discipline, namely, the transition-signal conceptual framework for the design of complex computation systems. Sutherland employs the 2-cycle, or non-return-to-zero (NRZ) signaling scheme. This is the most energy efficient and least time consuming signaling scheme. In transition signaling we do not distinguish between rising or falling edge of a signal. This means that, in effect, all responses to transition signals are edge triggered, and are triggered on both rising and falling edges. This results in the fact that the absolute state of control signals have no meaning. They are evaluated with respect to other related signals.

Transition signaling circuits must be symmetric with respect to the high and low states of control signals, since both rising and falling edges have the same meaning. This symmetry of transition signaling is highly desirable because it conforms with the symmetry of CMOS circuits.

If a sender and a receiver communicate using transition signaling, there will be two control wires and many data wires between them. The data wires carry conventional high or low states. The sender places a data value on the data wires *and then* produces a transition (rising or falling, we make no distinction) on the request control line to indicate that valid data are available. The receiver accepts data *and then* produces an acknowledge transition to indicate that the data have been accepted. The three events, data change, request and acknowledge always occur in cyclic order (though the lengths of the different cycles can be different).

2.5 Event Logic and Muller-C Elements

Control circuits for the above request-acknowledge protocols are built out of modules that form various logical combinations of events. 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 input OR the second input of the XOR will produce an output event. For more than two inputs, XOR generalizes to parity; parity circuit acts as a multiple input OR for events.

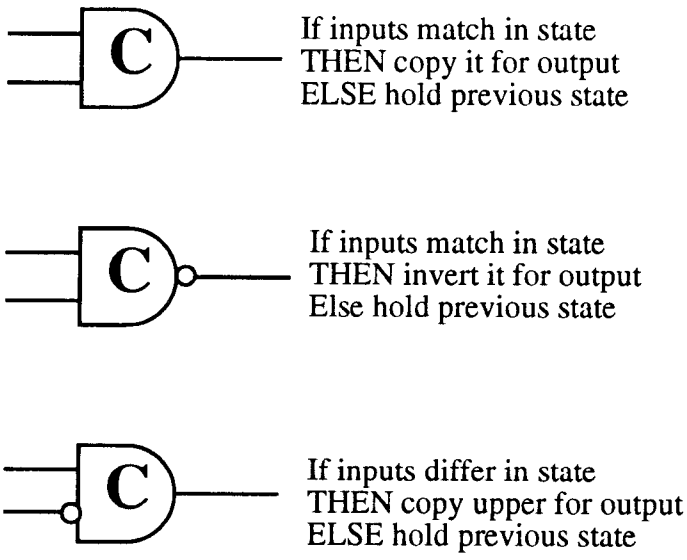

Figure 2.2 Muller-C elements for control of events.

The Muller-C element acts as the AND element for events. A two-input C-element implements the Boolean function C = AB + BC' + AC', where A and B are the two input signals, C' is the previous output signal, and C is the present output

signal. The C-element has the property that the output signal will change when both inputs are of the same level; otherwise the output stays unchanged. Thus only after an event takes place on both its inputs does an event occur at a C-element's output. The C-element is one of the basic units in the early approaches to designing speed-independent circuits. The C-element generalizes easily to three or more inputs requiring that all of them reach a new logical state before copying the new state as output. We use the standard AND logic symbol with a large C inside to represent a Muller C-element that implements a logical AND for transition events, as illustrated in Fig. 2.2.

Although the absolute state of a transition signal does not matter, its state relative to other related signals does. Thus it is sometimes important to invert transition signals. We use 'bubbles' on inputs or outputs of logic symbols to represent such inversions, as illustrated in Fig. 2.2. Every loop around which events flow must contain an odd number of inversions. Such loops are, in effect, oscillators whose oscillations are coordinated with those of other loops by the actions of C-elements or other modules at loop junctions.

2.5.1 SR-Latches vs. C-Elements

In classical logic design, the Boolean function of an Set/Reset (SR)-latch is written as Q(n) = S + R/Q(n-1). Here S and R are some combinational functions of input signals, Q(n-1) is the previous output signal and / is used to denote active-low signals. Thus S and R cannot be high at the same time, since the condition S = R = 1 would set the output Q undefined (a metastable condition) and represent a circuit hazard. This problem is usually circumvented by designing set-dominant (or reset-dominant) SR -latches in which the output is set high (or reset low) whenever S (or R)

is high. However, because of the assumed unbounded gate delays, there are situations in which the values of S and R cannot be predicted by the Boolean function. Hence the choice of set dominant SR-latch or reset dominant SR-latch becomes a function of gate delays.

Since logic delays are assumed to be finite but unbounded in speed-independent circuits, S and R may both become high with different gate delay assumptions. If we chose to abide by the orthodoxy of a pure speed-independent design, any logic implementation incorporating SR-latches cannot be truly speed-independent, as the mutual exclusion of S and R cannot be guaranteed through unbounded gate delays. C-elements do not have this problem and it has been proposed that speed-independent (and delay insensitive) circuits use only C-elements as memory elements. The functionality of an SR-latch is similar to a C-element with an inverter. Therefore C-elements can be used to replace SR-latches without any functional difference.

2.6 A 2-cycle FIFO

In Section 2.1, the basic concept of a event-driven pipeline was presented. Sutherland calls such a pipeline based on the transition-signaling framework - a *micropipeline.* It is from this name that we derive the "micro" part of the acronym MDF. The micro part of the name is appropriate, as a micropipeline consists of very simple circuitry, is useful in short lengths and is very suitable for layout in an integrated circuit. Also note that, when we look at general purpose computing structures, such a pipeline is used to implement the microinstructions as compared to a 'macro' instruction pipelined implementation.

Before we look at the actual structure of a micropipeline, let us understand the logic required to control such a pipeline. A string of Muller-C elements interspersed with inverters is the only logic required to control the pipeline. In fact, the third form of Muller-C element shown in Fig. 2.2, is the kind that would be perfect for such a type of control. We can then view this circuit by examining the state of each C-element relative to the states of the predecessor and successor C-elements. Remembering the behavior of the C-elements under discussion, one can see that the control of each stage follows a very simple stage rule:

IF predecessor and successor differ in state

THEN copy predecessor's state

ELSE hold present state.

Let us now try to understand how a pipeline without any processing, i.e. a FIFO, could be built with such simple control logic. In Fig. 2.3, we illustrate the basic structure of a micropipeline. A set of event-controlled storage registers in series serves as its data path while a string of Muller C-elements serves as its control. This event controlled storage element is required to respond to both the rising and falling transitions. This can be achieved by using two latches side by side, one controlled by a control wire called "capture", and the other by another wire called "pass", which are activated alternately. In fact, in Chapter 4 we will show that this event-controlled register is in fact modeled as discussed above.

The reason the registers have been arranged to be driven from one end while their control signals are being sensed from the other is because the control signals for the register must be amplified to drive all the switches in the many storage elements involved. Since the wires that carry control signals are long, there is always some

delay in controlling the register. The reader must note that, from here on, in all those figures that present both control and data signals, the control signals are denoted by dotted lines and the data path by either a thick line or a thin line.

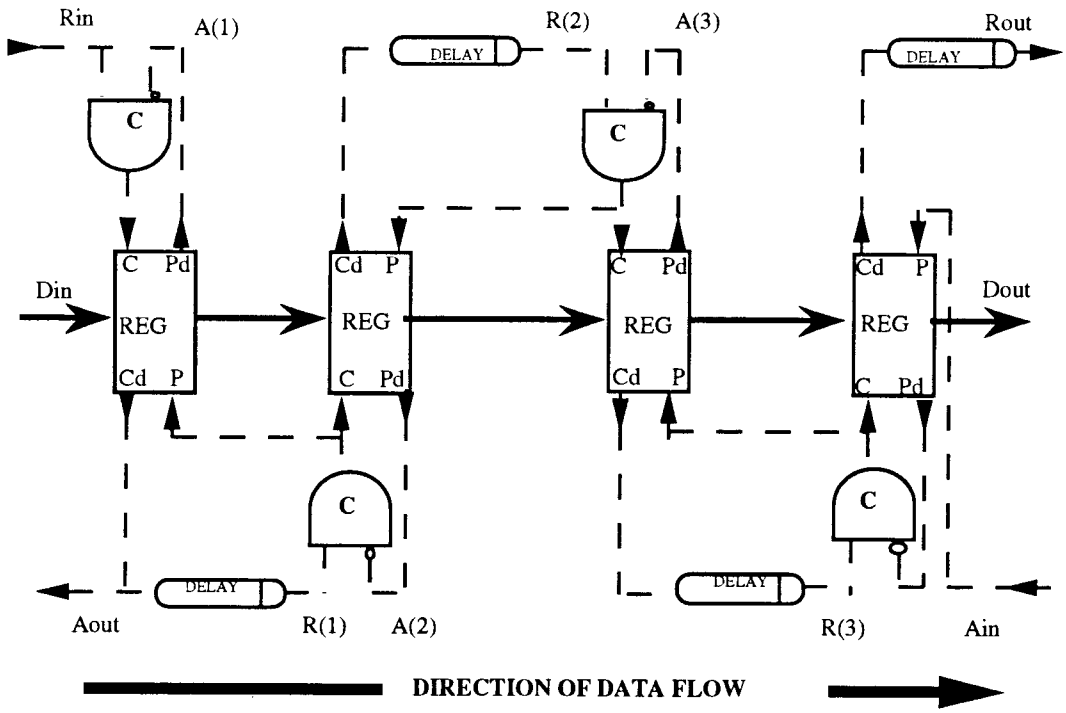

Figure 2.3 The Structure of a Micropipeline.

2.7 A 4-cycle FIFO

In a similar discussion, a 4-cycle FIFO can be developed. But before that we need to understand what kind of control logic would be required. Using the protocol discussed in Section 2.3 and the event logic discussed in Section 2.5, we can obtain a very simple circuit for the event control of a 4-cycle pipeline/FIFO, as shown in Fig. 2.4. This circuit is similar to the one developed by Meng.

With the help of this event control logic and the concepts developed in Section 2.6 on micropipelines, we developed a similar pipeline for a 4-cycle implementation. A significant difference between the storage elements used in the two FIFOs is that the 4-cycle storage elements will respond only on the rising transitions, or if the level of the request input is high. As will be obvious later in this discussion, these storage elements were in fact modeled in a similar fashion as discussed above.

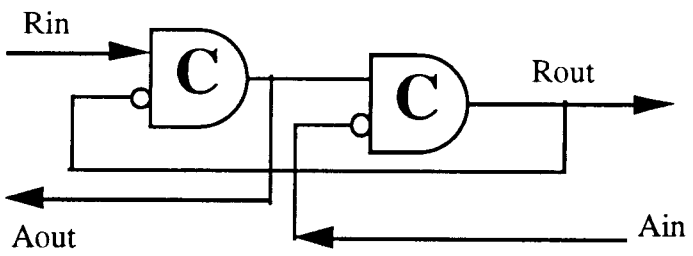

Figure 2.4 Event control for a 4-cycle pipeline.

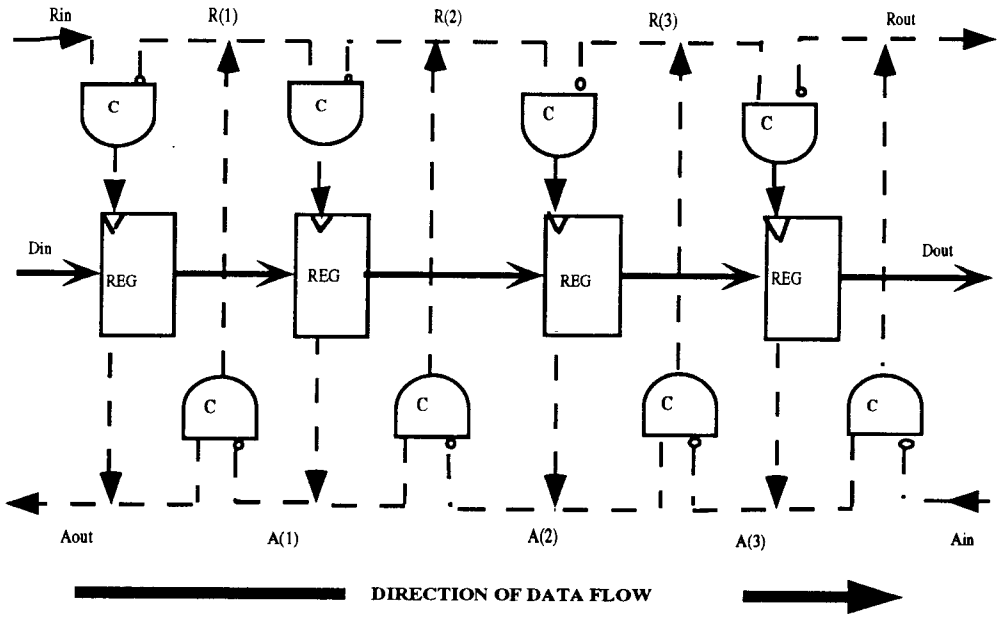

Figure 2.5 The Structure of a 4-cycle FIFO.

2.8 Some Interesting Observations

Naturally, data must propagate through an event-controlled pipeline faster than the control events propagate through its control. Sutherland assures this as follows. First the C-elements used in the control circuit are more complex than the storage element used in the data path, and hence inherently slower. Second, since each control stage of the control system must drive the many storage elements that hold a parallel word in each register, the control signals must be amplified to drive multiple loads. The amplification inevitably delays the control signals. Third, the layout of the circuit ensures that the zigzag path of the control signals has longer wires in it than those in the data path. If this can not be guaranteed, the request signal's arrival at the stage must be delayed by the computation time required by the current block.

Other suggestions are having the data path (i.e. the computation block) generate a completion signal. This completion signal can require considerable circuitry to implement. In Chapter 5, we will discuss a new logic family/topology at the transistor level that self-generates a completion signal.

The pipelines developed in the above 2-cycle and 4-cycle frameworks provide for a variety of pipeline processors. In fact, any micropipeline (we will now use it for both types of pipelines without copyright or loss of generality) with processing has combinatorial circuits placed between the storage registers. One can trade off the number of stages and the complexity of the intervening logic to obtain a suitable balance between latency and throughput rate. With less combinatorial logic between stages and more stages of storage, one obtains higher throughput rate at the cost of greater latency. This might entail greater hardware cost.

In this discussion, we are going to make an attempt to compare these two protocols and the issues that arise thereof. It is intuitive that there is a possibility of greater performance possible in the 2-cycle framework, but at what cost? But before we can assess the trade-offs therein we need to know what kind of control circuits would be required. We need to develop a higher level abstraction or method for specifying algorithms. This is the topic of discussion in the next chapter.

Chapter 3. DATA FLOW PROGRAM GRAPHS FOR DSP

3.0 Introduction

Digital signal processing (DSP) applications differ from general-purpose computation both in the nature of the algorithms and in the target hardware. The DSP algorithms usually involve a lesser degree of decision making. The target hardware is often dedicated to a particular algorithm, or at the most to a small class of algorithms, rather than being general purpose. They also have stiff cost/performance ratio requirements, especially in the case of real-time DSP. As mainstream computer science techniques can not be effectively employed, the DSP community usually designs its own microprocessors, languages, multiprocessor architectures and software.

DSP designers have been using "block-diagram languages" for high level specification. These languages have been nothing else but variations on the data flow representation of algorithms. Thus, a data-flow representation is very natural for the representation of DSP algorithms. Add to this the additional advantage of built in parallelism and concurrency, and we have a very simple and elegant method for specifying DSP algorithms.

In this chapter, we will try to highlight the major issues of concern. We will begin by explaining the basic concepts of data flow computing without regard to whether the the computation is general purpose, or application specific like DSP. We will then present a brief introduction to data flow program graphs. Finally, the issues concerning scheduling of data flow graphs for DSP will be briefly discussed.

3.1 The Data Flow Concept

For decades computers have been designed based on the control flow, more commonly known as the von Neumann model of computing. The corner stones of this philosophy are the program counters that are used to sequence the execution of instructions in a centralized control environment and the fact that data is passed between instructions via references to shared memory. Parallelism was restricted to the switching of a processor among separate processes and programmer-specified decomposition of a program into parallel instruction or data streams, to be processed by separate processors. Although in the last few years some work has been done in the field of parallel compilers to relieve the programmer of the tedium of partitioning the program into parallel processes, there are still many lacunae that need to be explored [10].

A major problem in a traditional multiprocessor system with a shared memory is memory interference. Given that a single processor usually tends to be memory bound, the addition of other processors attached to a shared memory provides only a limited gain (Amdahl's law!!). Local caches may have reduced the problem marginally, but it still remains a prickly problem, especially with the added consideration of memory integrity. Except for the advantages in the area of system availability, multiprocessor systems tend to have poorer cost/performance ratios than their uniprocessor system counterparts.

A second motivation for data flow architectures stems from properties of most current programming languages and the recognition that they mimic the underlying von Neumann computer organization. The organization is one of a passive storage (**variable**), a processing unit that performs state changes in the storage (**assignment**),

and a control unit that controls the state of the processing unit by sending it sequentially through a stream of instructions (**control flow**) [11,12].

Data flow advocates and programming language theorists argue that these three fundamental ideas are artificial, foreign to the way that programs should be expressed. These ideas contribute to programming complexity. Thus current programming languages were derived not from the outside-in (i.e. from the programmer's viewpoint), but from the inside-out (i.e. heavily influenced by the organization of earlier stored program machines).

The concept of data flow discards these basic principles by eliminating the idea of instruction streams and control flow, eliminating the concept of memory as a passive repository for program variables (an exception being associative memory), and providing a means of taking advantage of opportunities for parallel processing within programs without requiring explicit directions from the programmer.

3.2 A Data Flow Language

In a data flow computing environment, instructions are activated by the availability of data tokens (i.e. instructions/statements are data-driven). An instruction (or statement) is considered to be enabled (capable of being executed) when

(1) a datum exists on each of its input ports and

(2) no datum exists on its output port.

When an instruction is executed, the data on its input ports disappear and a result appears on its output port. Programs are represented by connecting instructions in a directed graph (i.e., connecting an instruction's output port to another instruction's

input port). Thus the order of instruction execution is controlled not by an instruction counter, but by the flow of data among instructions.

Control flow computers have synchronous computations performed using centralized control. Data flow computers are characterized by a passive examine stage. Instructions are examined to reveal the operand availability, upon which they are executed immediately if the functional units are available.

This data driven concept means asynchrony, which means that many instructions can be executed simultaneously and asynchronously. A high degree of implicit parallelism is expected in a data flow computer. Because there is no use of shared memory cells, data flow programs are free from side effects. In other words, a data flow operation is purely functional and produces no side effects such as the changes of a memory word. Operands are directly passed as tokens of values instead of as address variables. Data flow computations have no far-reaching effects. This locality of effect plus asynchrony and functionality makes them suitable for distributed implementation.

The language that will be discussed is basically a two-dimensional graphical language, which suits our purpose very well. The particular language to be examined is one proposed by Dennis [13]. Of course, there do exist data flow languages with the more familiar statement-oriented syntax, but they will not help illustrate the concept that will be discussed. In Dennis's language there are three major concepts:

3.2.1 Processing elements

A processing element is an operation that is enabled by the arrival of information on its input arcs and the absence of information on its output arc. The two categories of processing elements are *actors* and *links*. An actor is an operation with

one output arc and one or more input arcs. A LINK is an operation with one input arc and multiple output arcs [14].

3.2.2 Information

Information exists in the form of tokens (as we will see later, in our case it will be the transitions), which are transmitted over arcs and consumed and created by processing elements. The two basic types of information are *data values* (e.g. numerical values and *control or Boolean values* (e.g. true/false value). Note that at the circuit level the control values are also treated as data.

3.2.3 Arcs

An arc is an unidirectional path for information from one processing element to another. An arc can be either empty or contain a single token of information. The arc is the replacement for the traditional concepts of variables and storage. Because of the two classes of information, the arcs are classified as *data arcs* (denoted by solid lines here) and *control arcs* (denoted by dashed lines).

Fig. 3.1 illustrates the processing elements of the language. A **LINK (or FORK)** operation is enabled when a token appears on its single input arc, and all its output arcs are empty. On firing it distributes the input token to the output arcs. An **OPERATOR actor** normally has one or two input arcs. It is enabled for execution when data tokens are present on all input arcs (except for the MERGE actor), and its output arc is empty. It absorbs the input tokens, performs some function across these

values, and places the result data token on its output arc. Typical OPERATOR actors are addition, subtraction, multiplication, negation, square root, and so on.

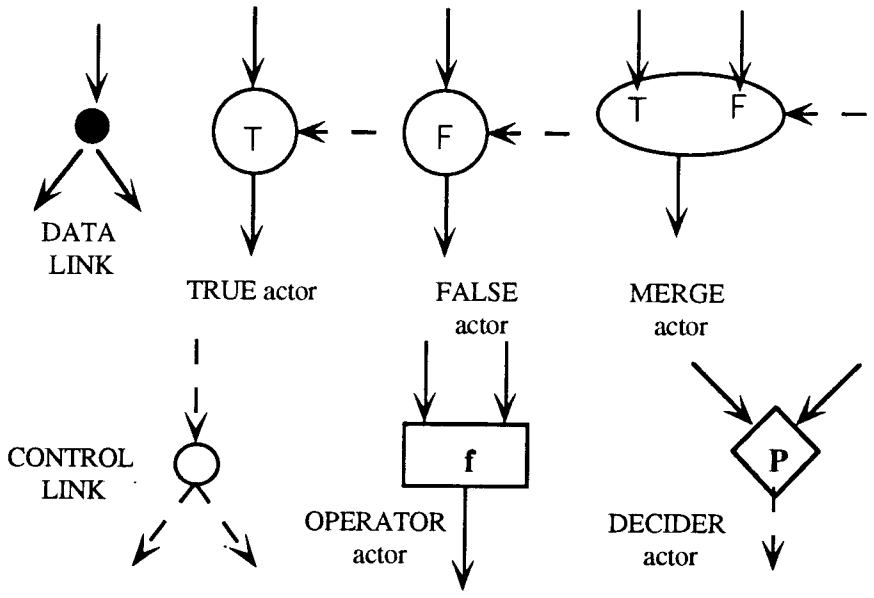

Figure 3.1 The Data Flow Links and Actors.

The **DECIDER actor (or PREDICATE, or JOIN)** is similar, except that it produces a Boolean or control result. It computes some PREDICATE across the data inputs, which produces a Boolean true or false value as a result. Typical PREDICATEs are equality, inequality, less than, and so on.

The remaining three actors have both data and control inputs. The **TRUE actor** is enabled when it has both a data token and a control token available as its inputs (and, its output arc is empty). Like all other elements, it absorbs its inputs when executed (fired). If the control value is true, the result is placed on the output arc; otherwise, no result is produced. Thus the TRUE actor will either pass its data input to the output arc, or it will simply absorb it. The **FALSE actor** is similar, except

that a false control value instead of a true one causes the data to be passed to the output arc.

The **MERGE actor** is an exception in that its execution does not cause all the input tokens to be destroyed, and it does not require all input tokens to be present to become enabled. It is enabled

- when a true control token is present and a data token is present on the data arc labeled T, or
- when a false control token is present and a data token is present on the data arc labeled F.
- In both cases, the output arc must be empty.

If the control input is true then the data token on the T input arc is transmitted to the output arc. These two input tokens are destroyed but the token on the F data input is not destroyed, if present. The opposite is valid if the control input is false.

3.2.4 Motivation and advantages

Graphical representations of data flow graphs offer many advantages [15]. Besides, the fact that a mental image of the behavior of a program due to the *data availability firing rule* is much easier to understand, data flow programs are easily *composable* into larger programs. Also data flow programs prescribe only essential *data dependencies*. It is very easy to attribute a formal meaning to a program by using a graph. This meaning can take the form of an operational definition (i.e. defines a permissible sequence of operations) or a functional one (i.e. describes a single function independent of the execution model).

3.3 Scheduling of Data Flow Programs for DSP

As discussed in the introduction to this chapter, we are interested in adapting data flow graphs for DSP. Edward Lee and D. G. Messerchmitt of Berkeley suggest that the data flow techniques of general purpose computing are too expensive for DSP and more powerful than required [16, 17, 18, 19].

In this section, we will deal with the scheduling strategies for data flow graphs. Scheduling is integral to the efficient exploitation of the inherent concurrency of a data flow graph. We will consider only non-preemptive scheduling.

Scheduling involves three basic tasks:

(1) Assigning actors to processors,

(2) Ordering the actors on each processor, and

(3) Specifying their firing time.

Every data flow implementation must have these three components. Implementations may differ by when (compile time or run time) they are implemented. Complexity of scheduling strategy also affects implementation.

Four classes of scheduling can be defined on the basis of which tasks are done when. They are:

(a) *Fully Dynamic* : Actors are scheduled at run time (i.e. they are assigned to a processor only when all the input operands are available).

(b) *Static allocation* : An actor is assigned to a processor at compile time, and a local run time scheduler invokes actors assigned to the processor.

(c) *Self-timed* : The order in which the actors will fire is determined by the compiler. At run time, each processor waits for data to be available for the next actor in its ordered list, and then fires that actor. This is analogous to self-timed circuits.

(d) *Fully Static* : The exact firing time of the actors, and their assignment and ordering information are all determined by the compiler. This is analogous to synchronous circuits.

Note that the boundaries between the different classes are not rigid. But it is interesting to note that as we go from strategy (a) to strategy (d), the degree of data dependency decreases, which means the strategies become less complex and cheaper. as they do not require special hardware support. In the case of self-timed scheduling, some synchronization primitives like simple handshake mechanisms are required. In fact, when extended to multiprocessor systems, the handshaking resembles Hoare's concept of *communicating sequential processes* [20, 21].

But the class of algorithms that can exploit the advantages of strategies (c) and (d) also reduces. Little or no data dependency, as well as comparable execution times of the various actors, are important features of self-timed scheduling and fully static scheduling. Although self-timed scheduling is better than fully static scheduling in tolerating some variations, it is not very good at that. This is not a severe problem in the case of signal processing algorithms and scientific computations [22, 23], like floating point arithmetic [24]. As a result, it can be observed that a self-timed schedule can be a very good option.

The compiler requires more information about the actors in order to construct close-to-optimal schedules. A solution is having the compiler construct a fully static suboptimal schedule and then discard the information that is not required. Static

allocation or assignment strategy requires only assignment information, while self-timed scheduling strategy requires both assignment and ordering information.

This gives rise to an important question - how are we to accomplish a fully static schedule for a self-timed scheduling strategy in the case of data dependencies like conditionals, recursion, and data-dependent iteration without losing out on the optimal strategies that would have been otherwise possible?

Considerable work has been done by Lee for synchronous data flow graphs, and by Meng for asynchronous data flow graphs at Berkeley under the guidance of Messerschmitt. For a detailed discussion, see the references in the bibliography. Here, we will present the different problems and discuss some of the suggested solutions and their effects.

3.3.1 Recurrences

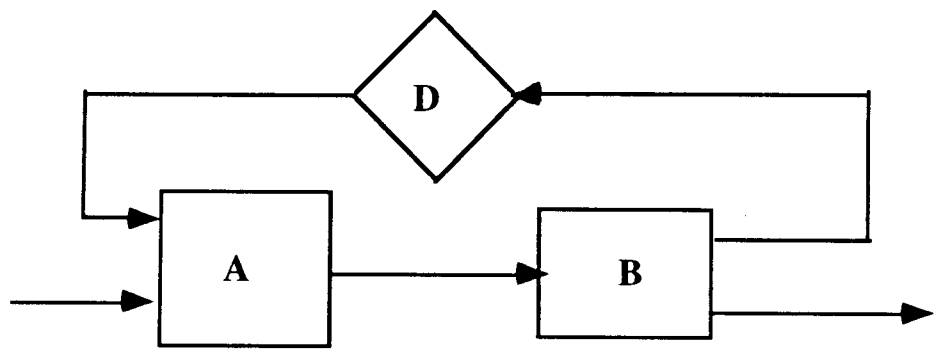

Figure 3.2 A data flow graph with a recurrence. Recurrences are expressed as directed loops and delays.

It is important to support recursion, or self-referential function calls. In fact, imperative languages implement recurrences and iteration in some combination. We will avoid the notion of "function calls". We can at least characterize some of the

recurrences as feedback paths in data flow program graphs. Data flow models for iteration will be examined in a later section.

A schematic of a data flow graph with a recurrence is shown in Fig. 3.2. This graph is assumed to fire repeatedly. The feedback path has a *delay* represented by a diamond. This can be implemented by an initial token on the arc (in Chapter 4 we will show the design of an actor that we call **INIT actor**). This delay is not a unit time delay but a logical delay or separator. A necessary (but not sufficient) condition for avoiding deadlock is to have at least one delay in a directed graph.

3.3.2 Conditionals

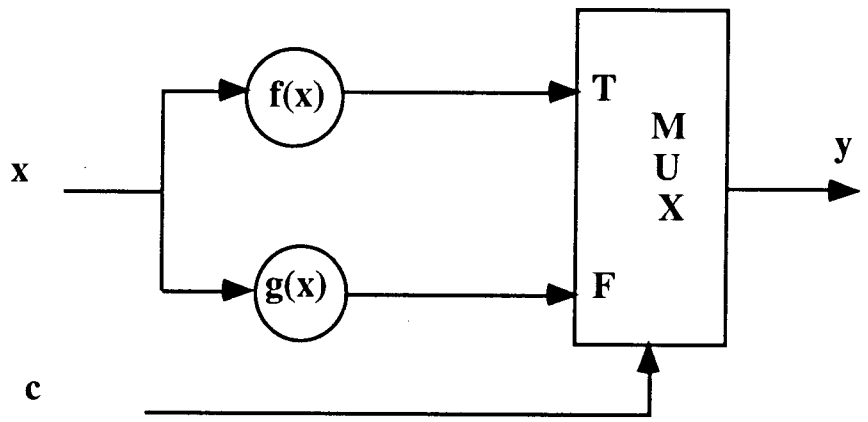

Figure 3.3 A data flow graph with conditional assignment. Both f(x) and g(x) are evaluated, and only one of them is selected.

Conditionals in data flow graphs are harder to describe and schedule statically. Conditionals are constructed within one large grain actor, and concurrency within such actors is difficult to exploit. Hence instead of *conditional evaluation* we use *conditional assignment* . We now require a slight variation on the MERGE actor that

we use. We have seen before that after a MERGE actor has fired it will not destroy all its input tokens. If we want to implement the functional expression (see Fig. 3.3)

$$y \leftarrow \text{if } (c) \text{ then } f(x) \text{ else } g(x),$$

we need a MERGE in which all the input tokens are destroyed each time the actor fires. Let us call this new kind of MERGE a **MUX actor**. Hence, both f(x) and g(x) are computed each time and only one of the results will be used. This is justifiable only when the functions are simple (deep pipelining), or for hard real-time applications when one of the two subgraphs is simple. Otherwise, the cost of unnecessary computation might be excessive.

An alternative for an if-then-else structure might be the use of a **SELECT actor** so that a token 'x' can be routed to one of the two functions, depending on the value 'c'. The appropriate function fires, and its value is selected by the MERGE actor. This has been schematically shown in Fig. 3.4.

3.3.3 Iterations

Iterations are of two types - manifest and data-dependent. Manifest iterations are of the type where we know the number of repetitions at compile time, and hence are independent of data. The others, as the name suggests, are data-dependent. They may be known before the iteration begins or after. Though manifest and data-dependent iterations pose different problems for the compiler, the resultant additional actors required are the same. They are used in increasingly complex configurations in order to achieve some degree of static scheduling.

The study of the problems with iterations required three more actors that were added to the four previously included. This will complete the set of data flow links

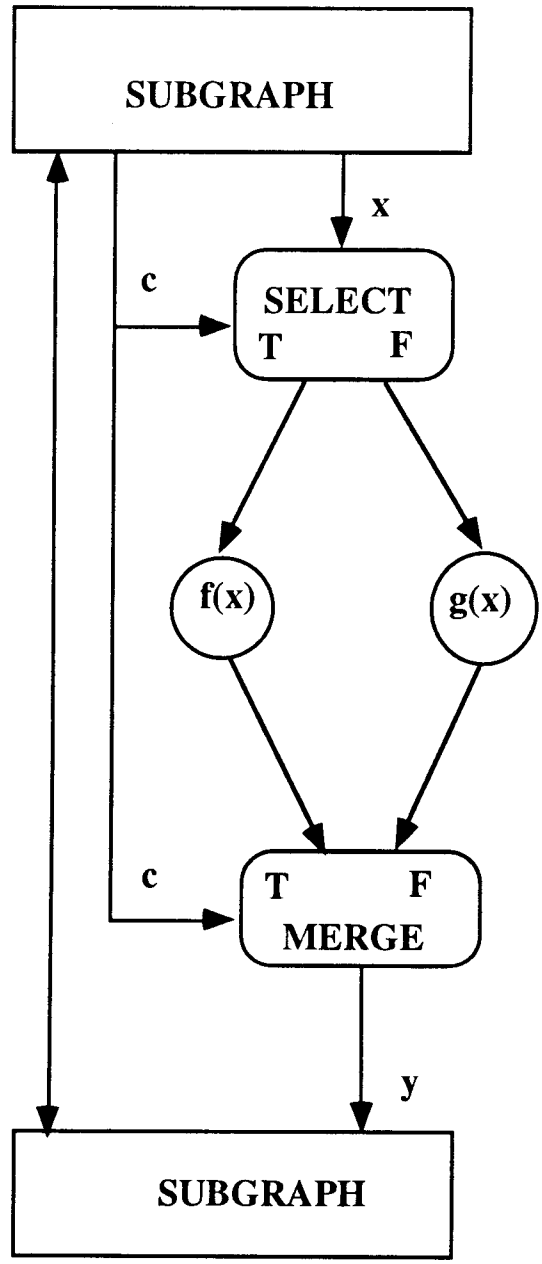

Figure 3.4 An alternative data flow graph for the expression y := if(c) then f(x) else g(x).

and actors for DSP applications. One of the actors that Lee suggests is called **Last of N**. It simply outputs the last of a series of N tokens, where N is a parameter of the

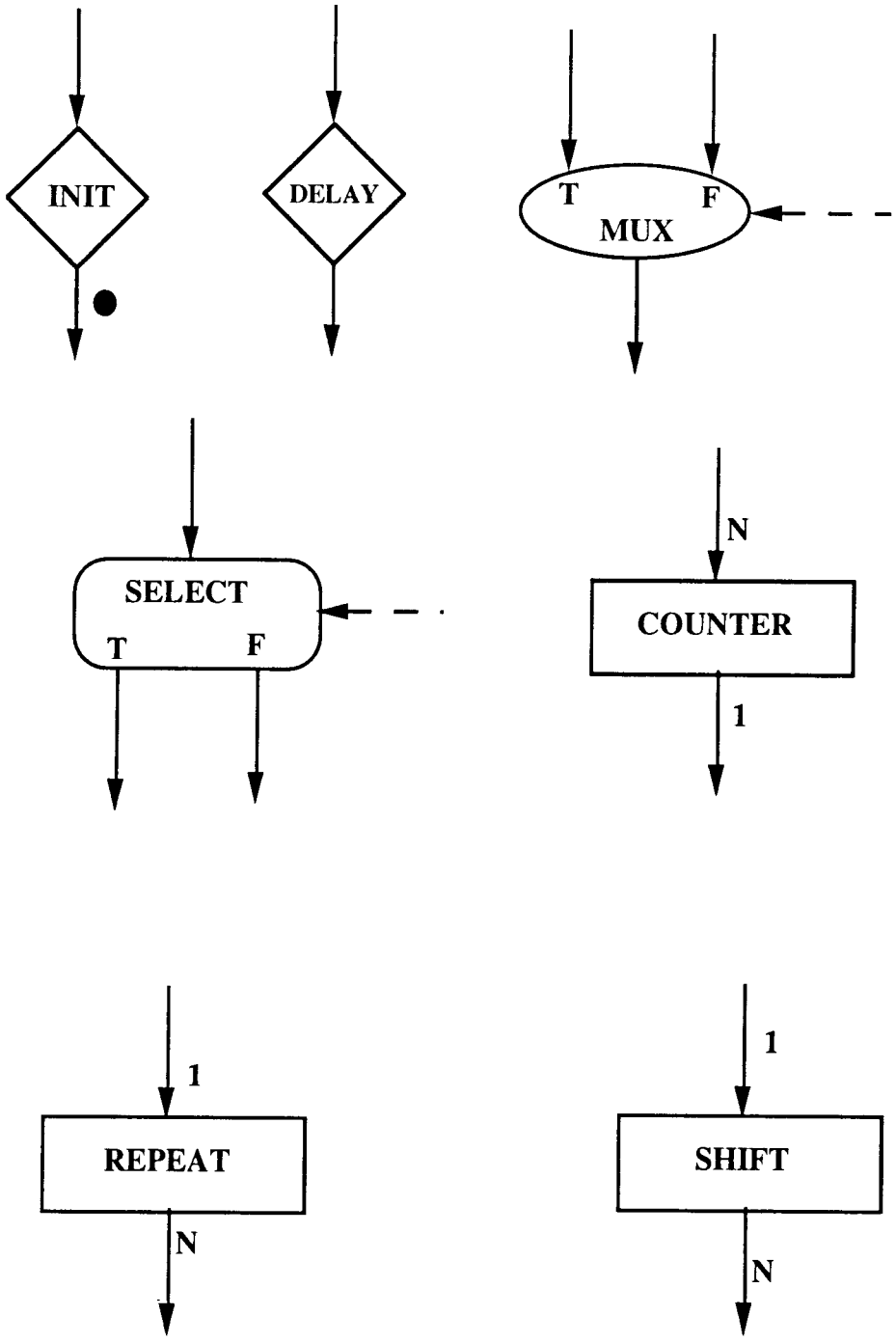

Figure 3.5 Data flow graphical representation of 6 additional actors.

actor. We call this, a little less euphemistically, a **COUNTER actor**. That is what it actually does - it counts N tokens and then lets only the last one through.

Another actor he suggests he calls **REPEAT actor**. It basically takes one input token and repeats it on the output N times. This can be implemented by using a FIFO and a COUNTER actor. A variation on this is required if we need to output the data in a serial fashion. We then require a parallel to serial shift register. We call this actor a **SHIFT actor**.

There are other actors that he suggests, like one that takes in one input token each time it fires, and outputs the last N tokens arrived. This can be built with a COUNTER actor and a N-deep buffer in a self-loop. Since this actor can be built by using other actors, it was not added to the list of links and actors.

Lee also suggest actors that he calls **UPSAMPLE** and **DOWNSAMPLE**. Both are variations on the COUNTER and REPEAT actors. Fig. 3.5 shows a graphical symbol for the 6 actors that have been added.

3.4 Some Interesting Observations

To the reader it would seem that, since all the additional actors can be constructed using the basic set, these actors are redundant. At this point, we will allude to a recent computer architecture related controversy - Complex Instruction Set Computers (CISC) versus Reduced Instruction Set Computers (RISC). The main motivation for RISC processors was not to minimize the size of the instruction set but to reduce it to the most relevant ones. Hence the authors feel that a more appropriate name would be Relevant Instruction Set Computers.

In the specific case of data flow graphs for DSP algorithms, we added these additional actors because of a high probability of their occurrence in such algorithms.

So, although the basic primitives have been increased the additional primitives are relevant and, hence, not unnecessary.

It is important to note that when we discuss the above actors, we are essentially interested in the control path, or, as we call it, the *distributed control path* (DCP). Although, the data path always exist along with most of the actors, its design is not under discussion here. Simple actors like the DECIDER actor and the OPERATOR have essentially similar control structure to a 'JOIN'. We call this structure a PREDICATE. Similarly, the control LINK and the data LINK, have essentially the same control structure and that of a 'FORK'. We will call this structure as a LINK.

3.5 Summary

In concluding the discussion on data flow graphs, we would like to emphasize that work still needs to be accomplished in terms of architectural development, compilers, etc. for self-timed data flow implementations to enable us to engage in fruitful activity as far as the design of programmable DSP processors. But it is definitely an interesting study to find out whether such an effort would pay off against using von Neumann based DSP processors.

A significant body of work has been developed in the last few years related to constructing a strong theoretical framework for the automatic design, development and testing of asynchronous circuits. In [25], A. P. W. Böhm presents a very good monograph on data flow computation. J. C. Ebergen [26] and E. Brunvand et. al. [27] discuss the automatic translation of programs into delay-insensitive circuits. A. J. Martin [28] investigates the compilation of communicating processes into delay-insensitive circuits.

T-A. Chu [29, 30, 31], T. Meng [32], C. J. Tan [33] discuss various methods of synthesizing self-timed circuits. Using trace theory for automatic verification of speed-independent circuits is discussed by D. L. Dill [34]. Dill along with Steven Novick of Stanford [35] discuss the same issue for self-timed circuits in Chapter 7 of Meng's book.

This chapter should enable the reader to understand the different actors required for an effective implementation of DSP algorithms. In the next chapter, we will discuss a hardware description language that allows us to define the functionality of the data path and the structure of the control path. Explicit specification of the data path is not required. Design energy can be concentrated in designing the DCP, and subsequently automating the design of the DCP.

Chapter 4. MODELING OF DIGITAL CIRCUITS USING VHDL

4.0 Introduction

Once we had developed an understanding of the kind of event control circuits that would be required, the task was to design them. But it was important not to get bogged down by details. VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) provided a very nice solution to this problem. In this chapter, we will explain how VHDL provides a very good method of describing the structure and function of ICs. Also, looking at the future requirements of this project, it is envisaged that some form of a translator of data flow program graphs into VHDL specifications will be needed. This form of automation would relieve the designer from worrying about all the details (for e.g., the design of the data path), especially in the infancy stage of a new design. These, of course, are the advantages that particularly apply to this project. In the next section, the basic advantages of VHDL will be specified.

This chapter is not intended as a tutorial on the many varied aspects of VHDL. References [36, 37, 38, 39, 40, 41] are provided in the bibliography that will accomplish that. Here we present the reader with some of the basic tenets of this new electronic circuit design philosophy and the approach that was taken by us in this project. This project was developed using View*logic*® VHDL [42].

4.1 Origins of VHDL

As has been stated above, VHDL is a language for describing digital electronic systems. It owes its origin to the United States Government's Very High

Speed Integrated Circuit (VHSIC) program, initiated in 1980. VHDL has now been adopted by the IEEE as a standard.

VHDL is designed to satisfy a number of needs in the design process. The main among them are:

(1) It allows the description of structure in an hierarchical form.

(2) It allows the specification of the function (or, as is said in VHDL jargon, **behavior**) of designs using familiar programming language forms.

(3) It allows a design to be simulated before being manufactured. It allows evaluation of the alternatives without expensive hardware prototyping.

4.2 Describing Structure

A digital electronic system can be described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the inputs. Fig. 4.1 shows a gate/flip-flop level circuit diagram of a double-edge triggered D flip-flop (DETDFF). Fig 4.2 shows the corresponding VHDL structural description. The module DETDFF has three inputs - D, CLK, RESET - and two outputs - Q, QB. Note that the outputs have been specified also as inputs because of the structure of the flip-flop. Using VHDL terminology, we call the module DETDFF a design **entity,** and the inputs and outputs are called **ports.**

One way of describing the function of a module is to describe how it is composed of sub-modules. Each of the sub-modules is an **instance** of some entity, such as PETDFF (positive-edge triggered flip-flop), NAND2 (2-input NAND gate), etc. The ports of the instances are connected using **signals.** This kind of a description is called a **structural** description. Note that each of the entities might also have a

structural description, as can be seen in the next chapter. On the other hand, the entities can all be **behavioral** descriptions (as discussed in the next section). A mix of both types of descriptions is also possible.

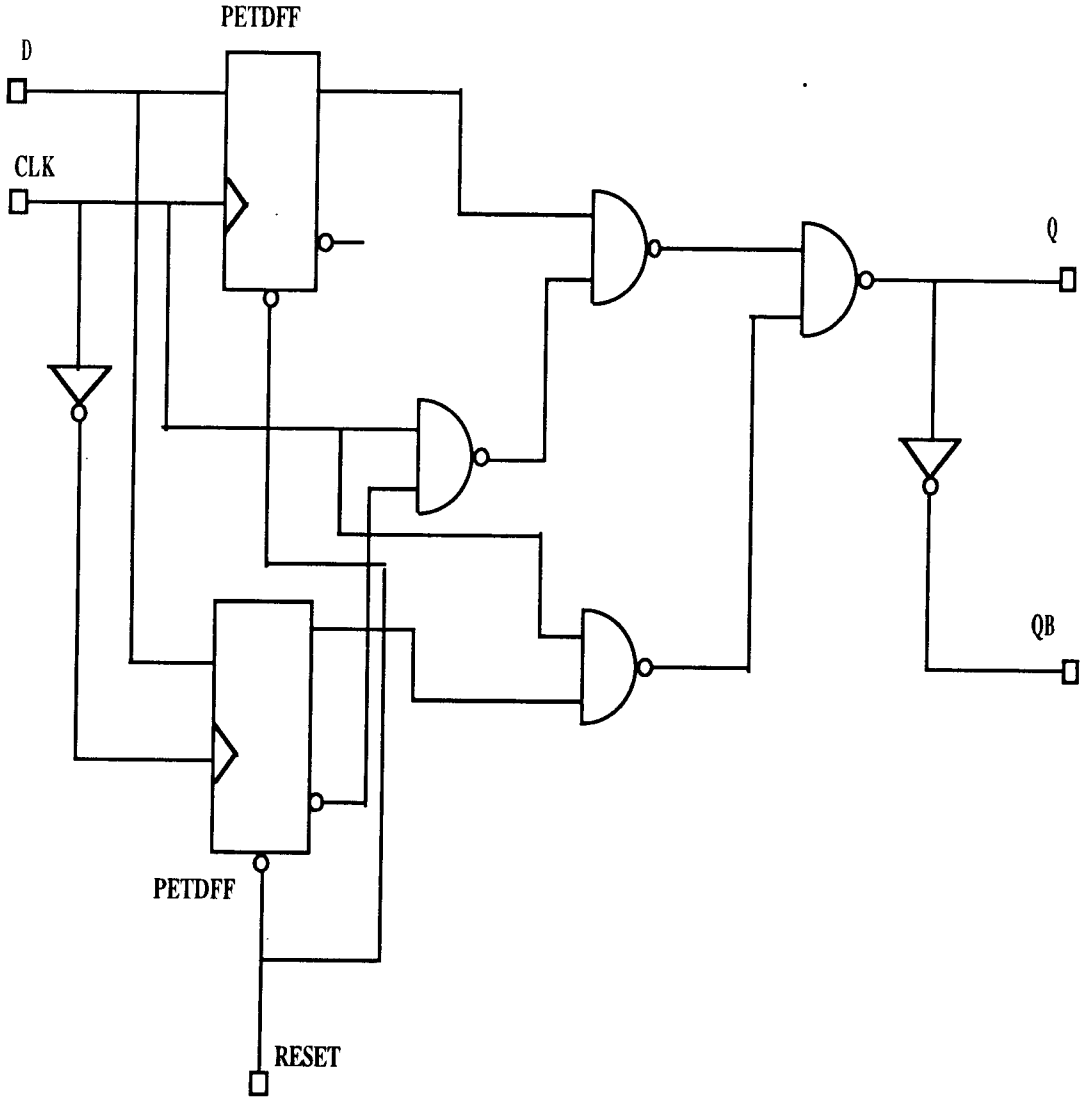

Figure 4.1 Gate level schematic of a double-edge triggered flip-flop.

```
-- VHDL structural model for a double-edge triggered D-FF with clear/reset

entity DSTRUCT is
       generic (DELAY : time := 5 ns);
       port (D, CLK, RESET: in vlbit;
                    Q, QB: inout vlbit);
       end DSTRUCT;

architecture STRUCTURAL of DSTRUCT is
       signal Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8: vlbit; -- internal signals
       signal DUMMY: vlbit;

-- component declaration for model NOT1
-- matches NOT1 entity declaration
       component NOT1
              generic ( TPLH: time := 1 ns;
                           TPHL: time := 1 ns);
              port ( signal QN: out vlbit;
                           signal A: in vlbit);
       end component;

       component NAND2
              generic (TPLH: time := 1 ns;
                           TPHL: time := 1 ns);
              port (QN: out vlbit;
                           A, B: in vlbit);
       end component;

       component PETDFF
              generic (DELAY : time := 2 ns);
              port (D, CLK, nCLR: in vlbit;
                           Q, QN: inout vlbit);
       end component;

-- 8 component instantiations define the model "structural".

       begin

U1: NOT1
              port map (Z1, CLK);
U2: PETDFF
              port map (D, CLK, RESET, Z2, DUMMY);
U3: PETDFF
              port map (D, Z1, RESET, Z3, Z4);
U4: NAND2
              port map (Z5, CLK, Z4);
U5: NAND2
              port map (Z6, Z2, Z5);
U6: NAND2
              port map (Z7, Z3, CLK);
U7: NAND2
              port map (Z8, Z6, Z7);
U8: NOT1
              port map (QB, Z8);
      Q <= Z8;

              end STRUCTURAL;
```

Figure 4.2 Structural description of a double-edge triggered flip-flop.

4.3 Describing function

In many cases, it is not appropriate to describe a module structurally. One such case is a module that is at the bottom of the hierarchy of some other structural description. For example, if you are designing a system using IC packages bought from a shop, you do not need to specify them structurally. In such cases, a description of the function performed by the module is required, without reference to its actual internal

```
-- VHDL behavioral model for a double-edge triggered D-FF with clear/reset

entity DETDFF is
          generic (DELAY : time := 2 ns);
          port(D, CLK, nCLR: in vlbit;
                    Q, QN: out vlbit);
          end DETDFF;

architecture BEHAVIOURAL of DETDFF is

          begin

P1: process
          begin
                    if nCLR = '0' then
                              Q <= '0' after 1 ns;
                              QN <= '1' after 2 ns;
                    end if;
                    wait until pchanging(nCLR) or pchanging(CLK);
                    if nCLR = '0' then
                                        Q <= '0' after 1 ns;
                                        QN <= '1' after 2 ns;
                    else
                      if nCLR = '1' then
                              if pchanging(CLK) then
                                        Q <= D after 1 ns;
                                        QN <= not(D) after 2 ns;
                              end if;
                              else
                                        Q <= 'X' after 1 ns;
                                        QN <= 'X' after 2 ns;
                              end if;
                    end if;
          end process;

          end BEHAVIOURAL;
```

Figure 4.3 Behavioral description of a double-edge triggered flip-flop

structure. Such a description is called a **functional** or **behavioral** description. Fig. 4.3 shows such a description of a double-edge triggered flip-flop.

It seems appropriate at this time to mention that all the gates (i.e. inverters, buffers, multi-input AND, NAND, OR, NOR, EX-OR, EX-NOR gates) have been described behaviorally. They are usually at the bottom of the hierarchy in any digital system design. Also, relatively complicated blocks like positive-edge triggered D flip-flop (PETDFF) and the Muller-C elements have been specified behaviorally.

In Chapter 6, we will see that complex data flow actors like ADDER PREDICATE have also been specified behaviorally. It is not important to have a structural description of an ADDER PREDICATE to effectively demonstrate the functional correctness and performance of the event control data flow LINKs and actors.

Note that the event control data flow links and actors have been described structurally. Although structural descriptions would suffice, schematic diagrams of the different circuits, save a few, have been presented in the next chapter for the uninitiated reader.

4.4 Discrete Event Time Model

Once the structure and behavior of a module have been specified, it is possible to simulate the module. This is done by simulating the passage of time in discrete time steps. At some simulation time, a module may be stimulated by the changing of the value at one of its input ports. The module reacts by executing the code and responding by changing the values on the output ports, if required, at a later simulation time. A *transaction* has been scheduled on the module. If the new value on the output port is different from the previous value, an *event* is said to have occurred.

This will result in other modules, whose inputs are connected to this output port to execute their respective codes.

The simulation starts with an *initialization phase* in which all the signals are given initial values, simulation time is set to zero, and each module's program is executed. This usually results in transactions being scheduled on the output ports at some later time.

This is followed by a *simulation cycle*. It is a two-stage cycle. In the first stage, the simulation time is advanced to the earliest transaction that has been scheduled. This transaction is executed, which may result in events occurring on some signals.

In the second stage of the simulation cycle, all modules that were affected by the events that happened in the first stage execute their respective program codes. Further transactions will be scheduled by these programs that will result in more events, and the cycle repeats until there are no more scheduled transactions. The simulation is then complete.

```
| This command file simulates a double edge-triggered D flip-flop using VHDL.
wave dstruct.wfm RESET CLK D Q QB
wfm RESET (0=0 50=0 50=1)
clock CLK 0 1 0 1 0 1 0 1
clock D  0 0 1 1 0 0 1 0
t RESET D CLK Q QB
stepsize 100
cycle 2
exit
```

Figure 4.4 A Viewlogic® VHDL Command File.

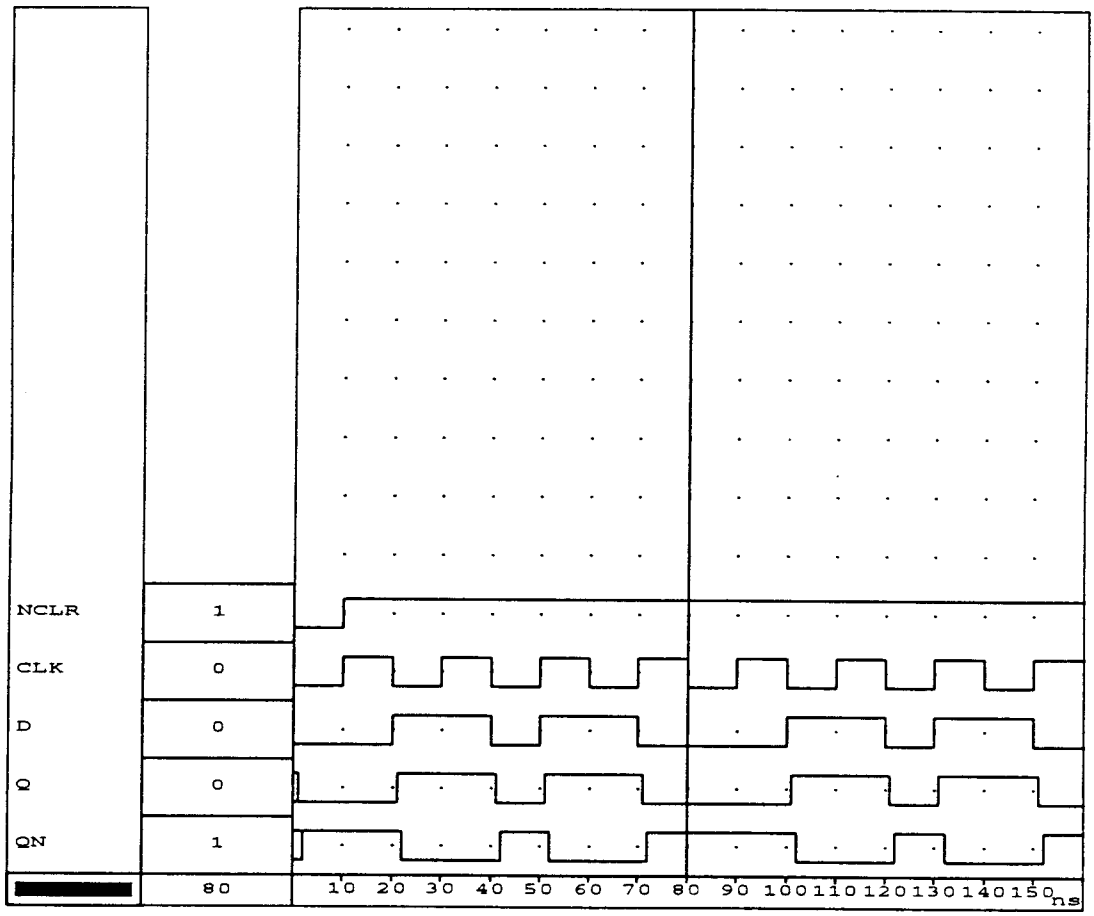

Figure 4.5 Waveforms generated by the simulation of a double-edge triggered flip-flop.

The simulation has gathered information about the changes in the system over the period of time. This is done by running the simulation under a *simulation monitor*.

The information can be stored in a trace file for later viewing and analysis. It could have an interactive feature.

The VHDL programs in this project were written using Viewlogic® VHDL. For the large part it is similar to the IEEE standard except for some minor variations. The simulation monitor in the Viewlogic® VHDL is called a *command file*. An example of the command file used to execute the structural and behavioral descriptions of the double-edge triggered flip-flop is shown in Fig. 4.4. Fig. 4.5 has the waveform that is a result of the simulation.

4.5 Summary

After identifying the need for a simulation tool that would let us concentrate on the design of the self-timed control circuits without getting mired in the details of implementation of the data path, we showed how VHDL provided a very good solution for this problem. In the next chapter the actual self-timed digital circuit designs of the various data flow links and actors (for both 2-cycle and 4-cycle request-acknowledge protocols) will be presented.

Chapter 5. 2-CYCLE AND 4-CYCLE SELF-TIMED IMPLEMENTATIONS OF THE DATA FLOW ACTORS

5.0 Introduction

Up until now, we presented the advantages of self-timed design and the two common request-acknowledge protocols that are used to design self-timed control circuits, which can be efficiently used for event-controlled pipelines. We then discussed the reasons why a data flow program graph provided a very good method for a high level specification of self-timed circuits. In particular, data flow graphs adapted for DSP applications were discussed in considerable detail.

In the last chapter, we discussed the advantages of VHDL as a modeling and simulation tool. In this chapter, we will present the self-timed digital circuit implementations of the various data flow links and actors for both the 2-cycle and 4-cycle request-acknowledge protocols.

As has been said before, the circuits that we will present are concerned with designing the *distributed control path*. We will concentrate on the issues related to the design of the control path. The data path has been characterized and discussed very well for quite some time now. But before we proceed to the design of the control path, we will briefly discuss the issues involved in the design of the data path.

5.1 Design of the Data Path

Before the actual designs are presented, the author feels that the uninitiated reader might find it interesting and rewarding to review some of the basic principles

of digital circuit design and the associated problems. There exist many good books. In the bibliography we suggest a few [43, 44, 45].

The reader will gauge from a brief perusal of the above literature that two main problems with asynchronous design are mentioned. The first one is said to have occurred when the output of a combinational network is in error due to the changing of signal values on more than one line in the network, more commonly known as a *hazard.* This behavior is transient in a feedback-free network. In [46, 47] the reader will find a very good discussion on the design of asynchronous circuits with bounded gate delays. Assuming bounded gate delays, allowing only one of the primary inputs to change at a time and including all the prime implicants (i.e. in the fundamental mode), guarantees that temporarily erroneous outputs will not occur. In sequential circuits, race conditions can lead to steady-state errors. Under the assumption of bounded gate delays, this problem can be solved by including redundant states and making the delay elements long enough for all transients to settle down before transmitting a change of state.

D. B. Armstrong et. al. [48] published one of the first works on design of asynchronous circuits assuming unbounded gate delays. More recently, C. H. Lau et. al. [49] discussed the design of asynchronous circuits using a data flow approach. Both used data detectors or spacers. This method reduces the hardware efficiency to less than 50%.

Another difficulty often mentioned of asynchronous design is the performance uncertainty due to the problem of *metastability.* An example is arbiters that grant mutually exclusive access to a particular resource. Simultaneous requests can drive the arbiter into metastability, where the output is nondeterministic for an undefined period of time. In fact, any cross-coupled inverter pair has the potential to exhibit metastability. In synchronous systems, the problem of metastabilty is solved by

designing the clock period such that the latching signal always falls behind the data transition. There exists a large body of work that has discussed this problem as related to asynchronous system design. The solution is to derive the latching signal from the data signal so that latching will definitely take place after the data is valid.

At this point, we would like to draw the reader's attention to the micropipeline discussed in Chapter 2. We mentioned then that the 'delay' element in the path of the request signal is to restrict the latching signal (i.e. the request signal) from arriving at the next stage before the data is valid. When designing systems in VLSI, it is impossible to guarantee with any degree of uncertainty that such delay would hold within tolerable limits even after process variations are accounted for.

This leads us to a discussion on the transistor level topologies that are employed. A logic family that is used quite often is Dynamic Cascode Voltage Switch Logic (DCVSL)[50]. Theresa Meng uses this logic family. She suggests that to generate a completion signal from the data that could be tied into the request signal, we use a level of OR gates, one for each data pair. We now have the completion signal of each data pair. To generate the completion signal for the entire data bus, a two level AND gate implementation would suffice, provided the precharge delay is approximately the same for all data bits. If this prerequisite cannot be guaranteed, then a tree of C-elements will be required - an expensive solution !!

S. L. Lu et. al. [51, 52, 53] discusses a novel topology. This topology is called Enable/Disable CMOS Differential Logic (ECDL). Sutherland uses static CMOS for the implementation of the data path. ECDL avoids the disadvantage of charge sharing of DCVSL. It also automatically generates a completion signal avoiding delay calculations required for static CMOS implementations. It also lends itself very naturally to mapping from a high-level specification.

We will leave this discussion at this point, except for a brief mention later, as research is currently being undertaken to automatically synthesize Boolean equations into ECDL circuits.

5.2 Pipeline and Non-Pipeline Interconnections

Pipelining increases system throughput by introducing sample delays (i.e. registers in the data path). This directly translates into greater hardware cost. Also there are situations when the two computation blocks operate sequentially. No intermediate sample delay is now required. This gives rise to the need of a non-pipeline interconnection, which will allow only one block to be active at time, and not half the blocks. It can later be observed that a non-pipelined interconnection does not require a register.

5.2.1 2-cycle interconnections

The basic 2-cycle pipeline interconnection is similar to the Muller-C element used in the 2-cycle micropipeline. The non-pipeline interconnection used to connect two sequential blocks can be accomplished by connecting R_{in} to R_{out} and A_{in} to A_{out}. This is possible because ECDL does not require a specific precharge phase as compared to DCVSL. As we will see in the next sub-section, the above fact will affect the design of the 4-cycle non-pipelined interconnection.

We do not present either specific circuit schematics or simulations of the above two circuits, as their performance can be intuitively understood.

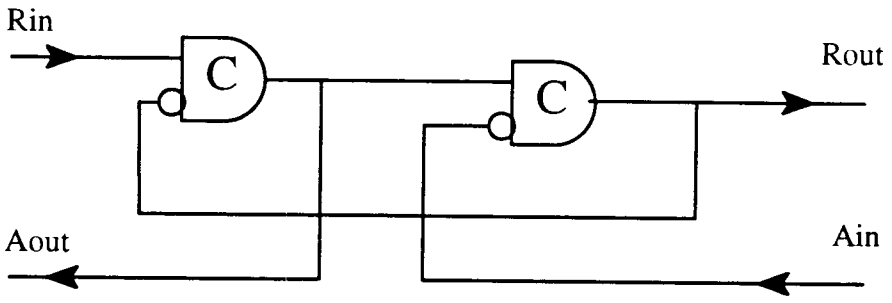

Figure 5.1 4-cycle pipeline interconnection.

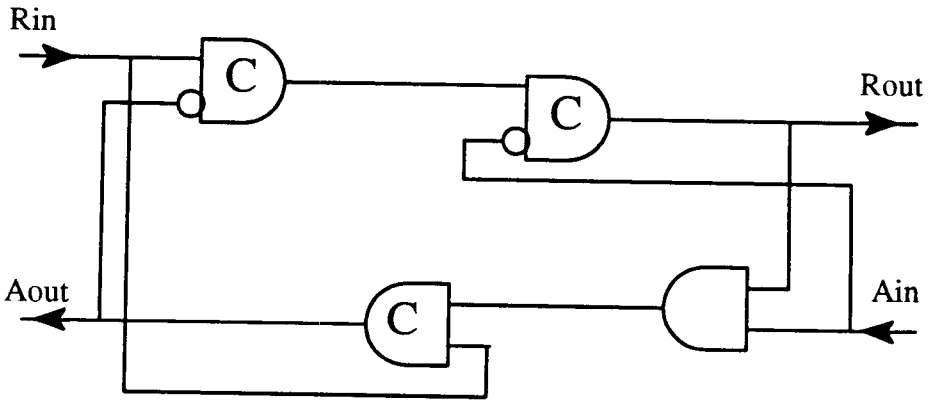

Figure 5.2 4-cycle non-pipeline interconnection

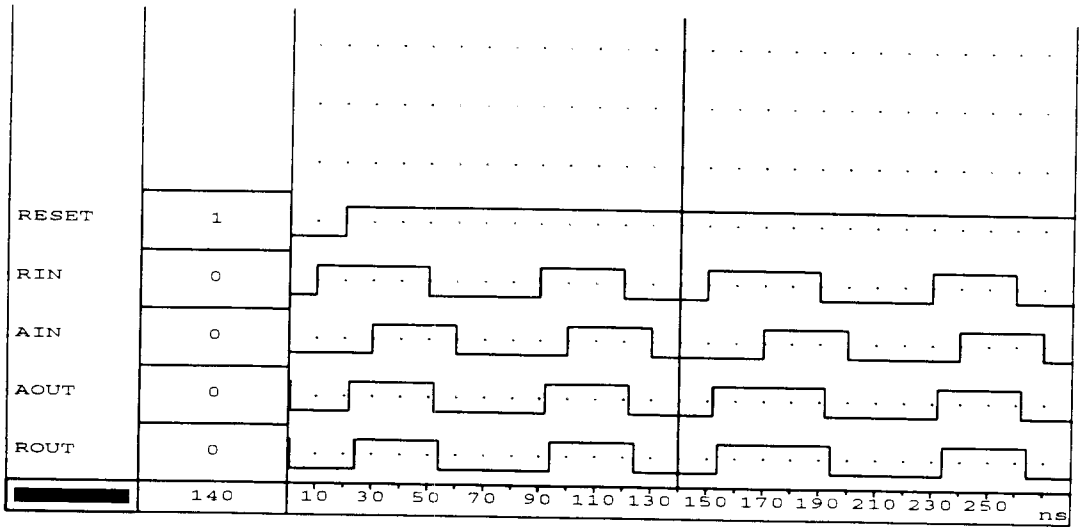

Figure 5.3 Simulation result of the 4-cycle pipeline interconnection.

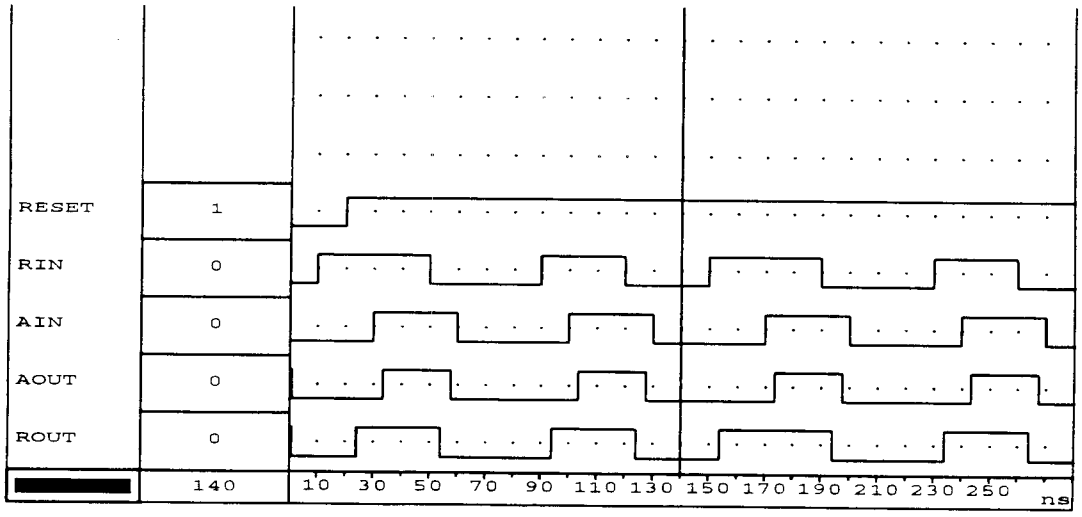

Figure 5.4 Simulation result of the non-pipeline interconnection circuit.

5.2.2 4-cycle interconnections

The 4-cycle pipelined interconnection circuit is similar to the event control of the 4-cycle micropipeline. It was first shown in Figure 2.4 and has been reproduced in Figure 5.1 for purposes of continuity. The non-pipelined interconnection is illustrated in Figure 5.2. A simpler circuit would have been created by connecting R_{in} to R_{out} and A_{in} to A_{out}. However, the circuit in Figure 5.2 allows a form of concurrency by allowing block A to start computing the next datum (i.e. precharge) while block B is being reset. In all designs of 4-cycle non-pipelined interconnections, we will choose the option that offers greater concurrency.

The simulation results of the 4-cycle pipelined and non-pipelined interconnections are shown in Figure 5.3 and Figure 5.4, respectively. Notice in Figure 5.3 that the A_{out} signal follows R_{in} before R_{out} does, allowing for R_{in} to reset. This leads to concurrency. The non-pipelined interconnection does not allow A_{out} to change until A_{in} has changed, leading to the fact that only one block can be active at a time.

It can then be concluded that a pipeline interconnection requires a register (of the type discussed in Chapter 2) to latch the next datum. Note that, in all future pipelined interconnection circuits, the registers have not been included, but are implicit in their presence. On the other hand, the non-pipelined interconnection does not allow another datum to be operated upon until the next stage is free to accept the result. Hence, no register is required.

In the next section, we begin our discussion on the 2-cycle and 4-cycle implementations of the various data flow links and actors.

5.3 The Data Flow Links and Actors

The complete set of data flow links and actors have been designed for both 2-cycle and 4-cycle implementations. For each of the protocols, both pipelined and non-pipelined interconnections have been designed. This results in an impressive array of designs, which for the purposes of brevity can not all be included in this document. In the following sections, we present the pipelined implementations for the two protocols. For additional details about the non-pipelined interconnections, the interested reader is requested to contact the author.

At the end of this chapter, we tabulate the approximate transistor counts and approximate delays for all the different implementations of all the actors.

To be concise actors that show the same properties have been grouped together because their event control circuits do not differ significantly. Example are the TRUE actor and the FALSE actor. The only difference is the presence or absence of an inverter in the path of the controlling Boolean signal. Another case is the REPEAT and SHIFT actors. Here the data path changes complexion. In the case of the REPEAT actor, the output data is parallel. On the other hand, for the SHIFT actor the output data becomes serial.

Both the control and data LINK have been grouped as a single LINK entity since their control path is similar. Also, in the case of the OPERATOR actor and the PREDICATE actor, since the control resembles that of a JOIN, we have a single entity called a PREDICATE actor. The INIT actor and the basic DELAY actor (or the IDENTITY actor) have been grouped together as an INIT actor. The only difference is whether one of the active elements was SET or not at the time of system reset.

We thus have the following links and actors - (i) LINK, (ii) PREDICATE, (iii) TRUE actor, (iv) MERGE actor, (v) SELECT actor, (vi) MUX actor, (vii) INIT actor, (viii) COUNTER actor, and (ix) REPEAT actor.

Also in the context of the 2-cycle actors, since the basic protocol does not differentiate between a rising transition and a falling transition, some interesting design considerations arise. We introduce two new terms - *in-sync* and *out-of-sync*. In-sync signifies that the transitions are of the same type: rising or falling. Out-of-sync denotes that the transitions are of opposite types. These impose different criteria on the designs, as will be clarified in subsequent sections.

In fact, the assumption that the request inputs are in-sync eases the design complexity. But is this assumption valid? On study of many general data flow graphs, it was concluded that if the actor follows the basic static firing rule - fire only when ALL input tokens are available and NO output token is available - then ALL input tokens are consumed and ONE output token is produced. The exceptions are the TRUE, FALSE and MERGE actors. As a result, it can be observed later that these two actors have the most complex designs.

Also the COUNTER and REPEAT actors do not strictly adhere to the static firing rule. But since there is only one request input, the problem is not as complex as it seems. This places different restrictions. The reader will find the solutions interesting. In [54, 55, 56], the reader has access to some quick references to the material that will be presented in the rest of this chapter.

5.3.1 Self-timed implementation of the data flow LINK

An simple 2-cycle implementation of the LINK is provided in Figure 5.5. The assumption that the two acknowledge input signals are in-sync is valid since

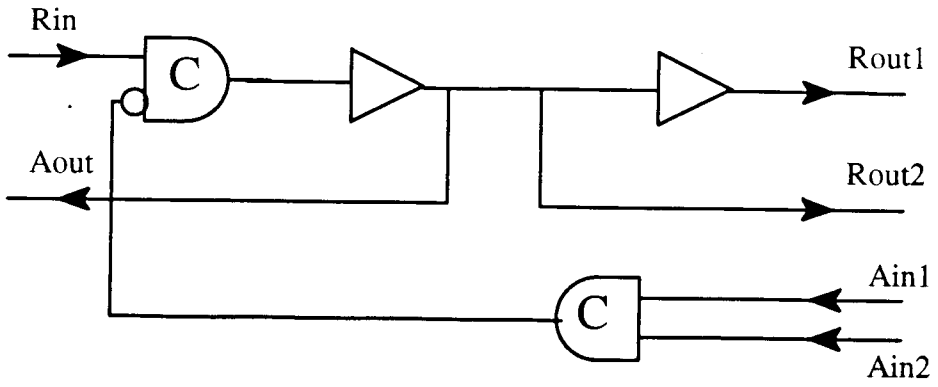

Figure 5.5 2-cycle implementation of a Data Flow LINK.

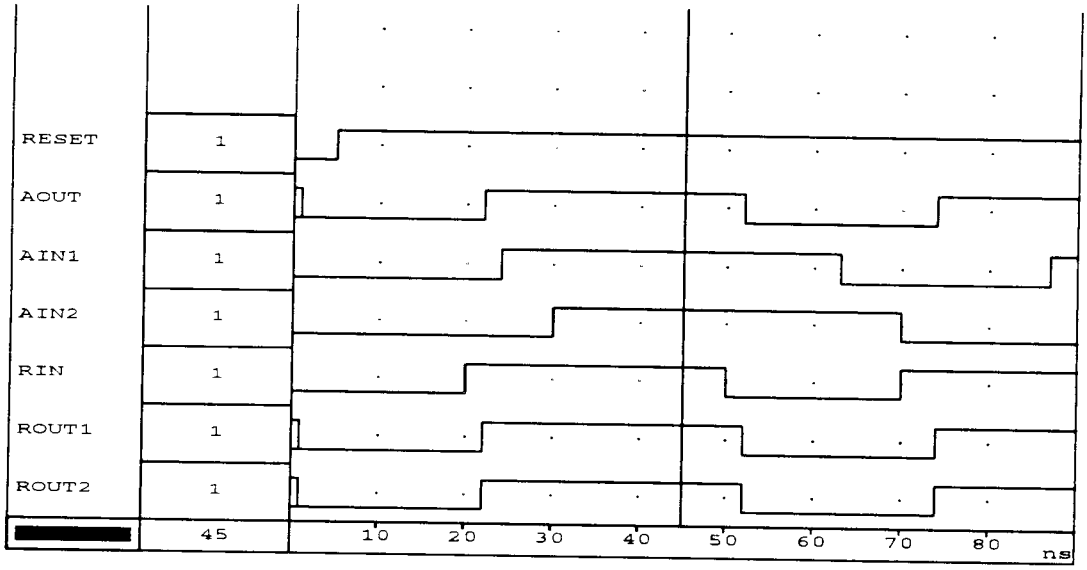

Figure 5.6 Simulation result of a 2-cycle Data Flow LINK.

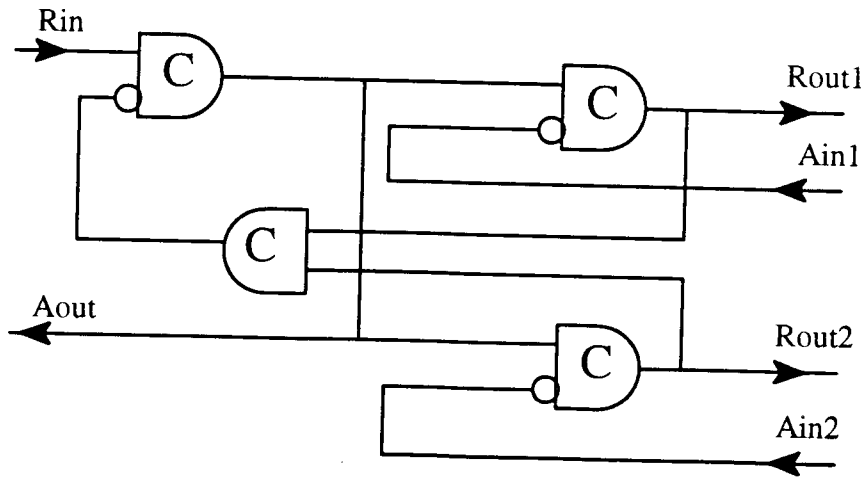

Figure 5.7 4-cycle implementation of a Data Flow LINK

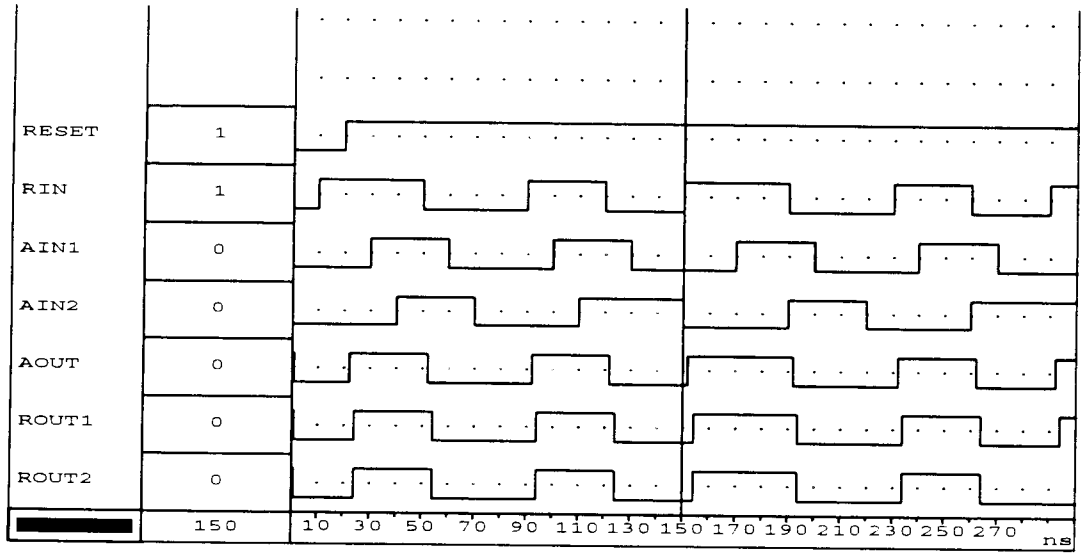

Figure 5.8 Simulation result of a 4-cycle Data Flow LINK

the outgoing request signals are always in-sync. Thus, by using a Muller-C element, we ensure that only when the two output tokens have been acknowledged will the next token, which may have already arrived (remember A_{out} followed R_{in} immediately), be consumed. This avoids the run-away condition. The simulation result is illustrated in Figure 5.6.

We turn our attention to the 4-cycle pipelined implementation of the LINK illustrated in Figure 5.7. The reader can observe that its design is a logical extension of the 4-cycle pipeline control primitive of Figure 5.1. The only addition is the Muller-C element in the feedback path. Figure 5.8 shows the simulation result.

5.3.2 Self-timed implementation of the data flow PREDICATE

Figure 5.9 illustrates the 2-cycle implementation of the data flow PREDICATE actor. The design is logical and should be clear to the reader. The assumption that the two request inputs are in-sync is valid because of the static firing rule and because on system reset all signals start in a common state (either low or high, although in our simulations they start low). The corresponding simulation result is illustrated in Figure 5.10.

The 4-cycle implementation and the corresponding simulation result are depicted in Figure 5.11 and Figure 5.12, respectively. Again the design is similar to that of the 2-cycle implementation except for the additional Muller-C element and a difference in the feedback path. This change allows all the signals to return to a zero state.

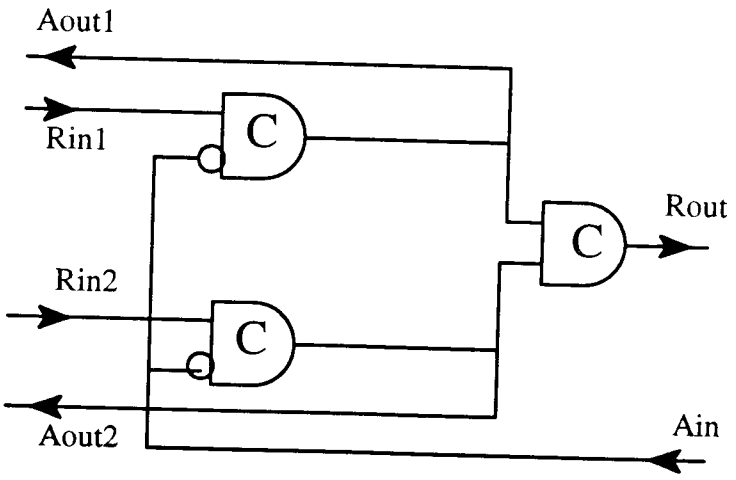

Figure 5.9 2-cycle implementation of a Data Flow PREDICATE actor.

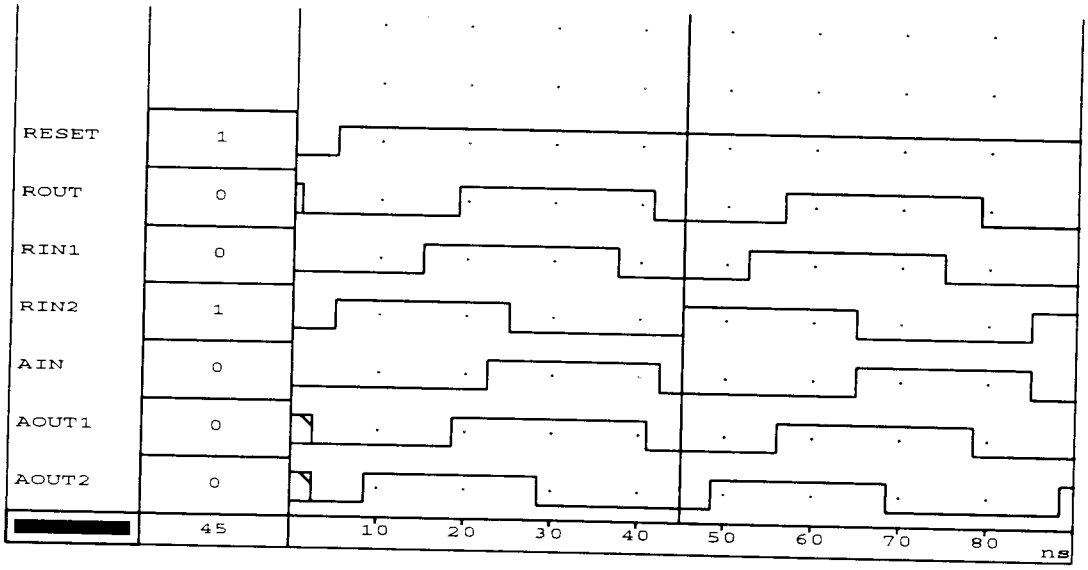

Figure 5.10 Simulation result of a 2-cycle Data Flow PREDICATE actor.

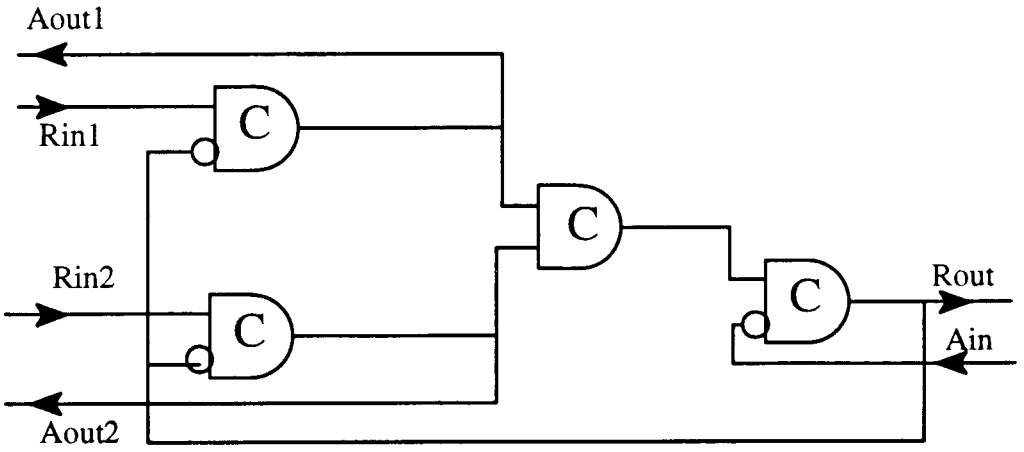

Figure 5.11 4-cycle implementation of a Data Flow PREDICATE actor.

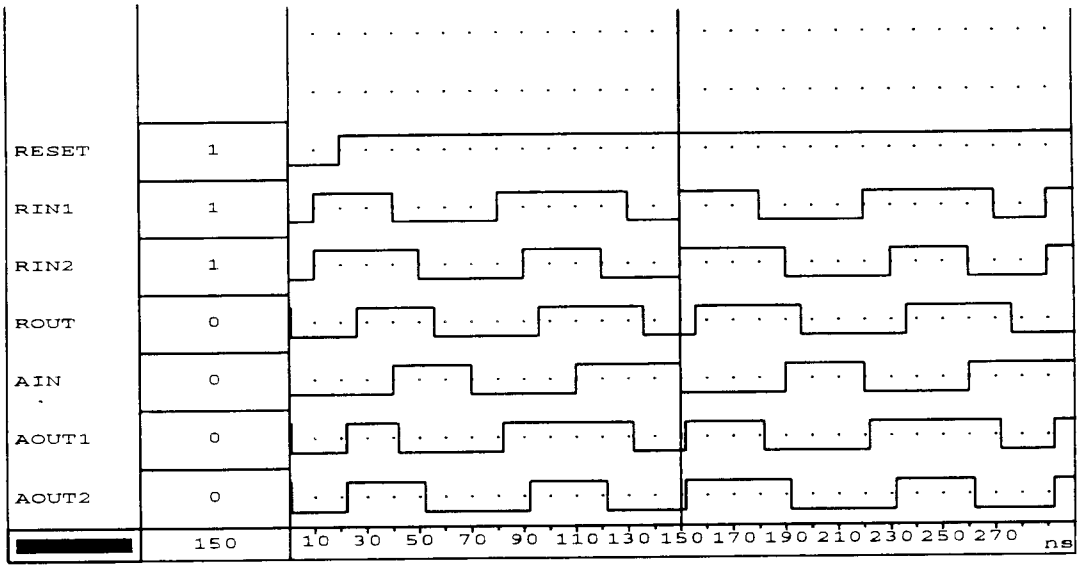

Figure 5.12 Simulation result of a 4-cycle Data Flow PREDICATE actor.

5.3.3 Self-timed implementation of the data flow TRUE actor

In the previous two sections, we discussed the 2-cycle and 4-cycle implementations of the LINK and PREDICATE actor. The similarity between the 2-cycle and 4-cycle implementations was interesting. We now discuss a more complex actor. Because the TRUE gate has a different firing rule, we will see a marked difference between the 2-cycle and 4-cycle implementations.

The firing rule says that once both the input tokens - the data token and the control token - have arrived and no output token is present, an output token will be created, depending on the Boolean value of the control input after consuming the input tokens. As a result, although we can assume that the two input requests will arrive in-sync (as they are both consumed each time), the same assumption is not true on the request input and the acknowledge input.

Let us first discuss the schematic of the 2-cycle implementation illustrated in Figure 5.13. Here the initial Muller-C structure is similar to that of the LINK. The double-edge triggered flip-flop together with the EX-OR gate is a structure that is used to allow us to control the generation of R_{out}, depending on the Boolean value of T. Another such structure is used to create A_{out} even if no A_{in} is available (because R_{out} was not generated).

An interesting property of an EX-OR gate, is that if one of the inputs is held at logic '0', then the other input (in our case the QB output of the double-edge triggered flip-flop) is passed without any change to the output of the EX-OR gate, which is connected to the D input of the flip-flop (i.e. a toggle configuration has been created).

On the other hand, if one of the inputs of a 2-input EX-OR gate is held at logic '1', then the output is the inverted logic of the other input. This results in the QB output of the flip-flop being inverted. The logical equivalent of the output Q of the

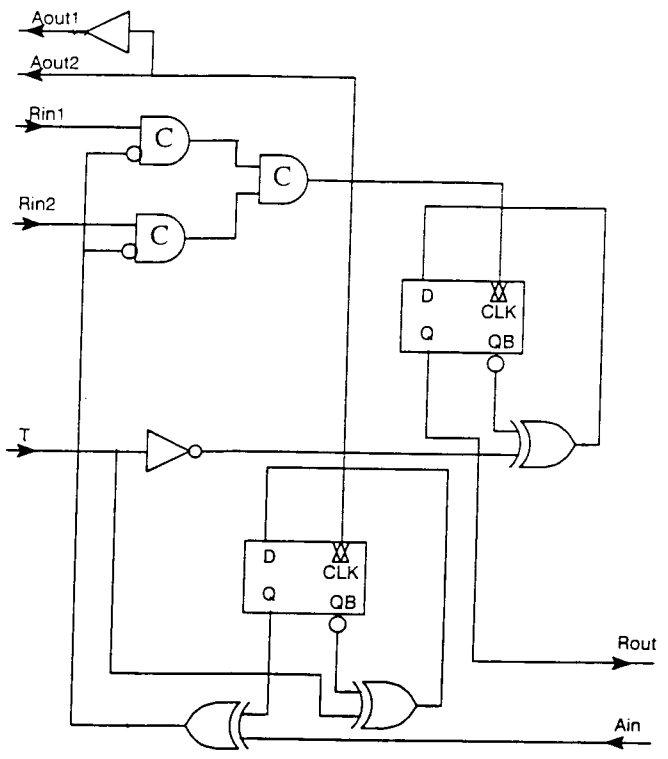

Figure 5.13 2-cycle implementation of the Data Flow TRUE actor.

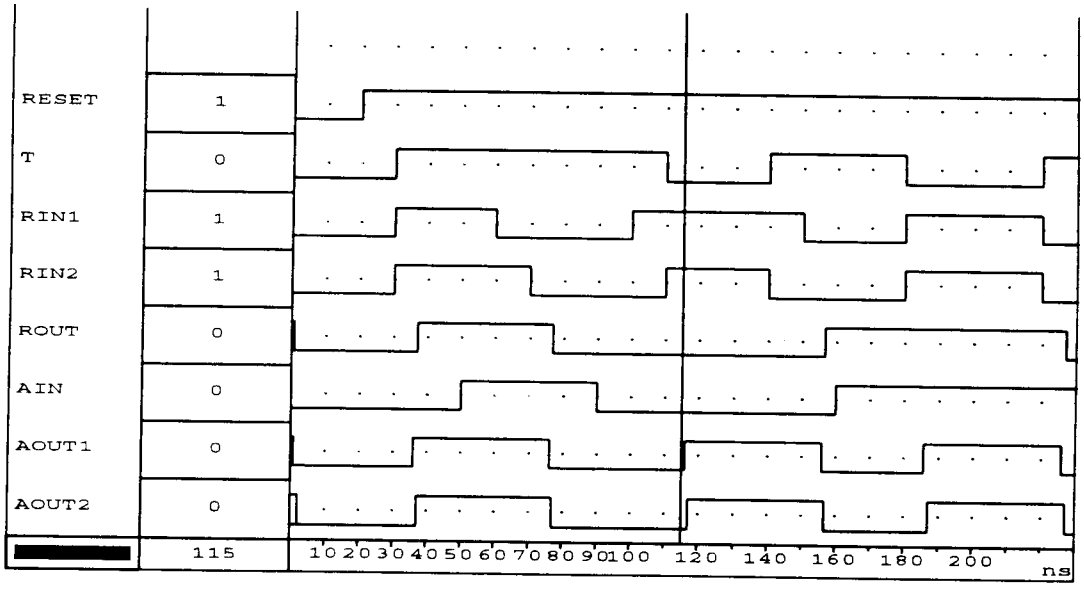

Figure 5.14 Simulation result of a 2-cycle Data Flow TRUE actor.

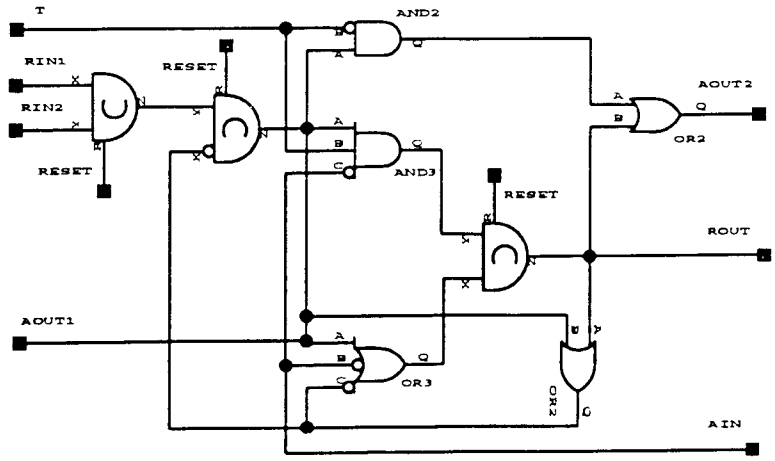

Figure. 5.15 4-cycle implementation of a Data Flow TRUE actor.

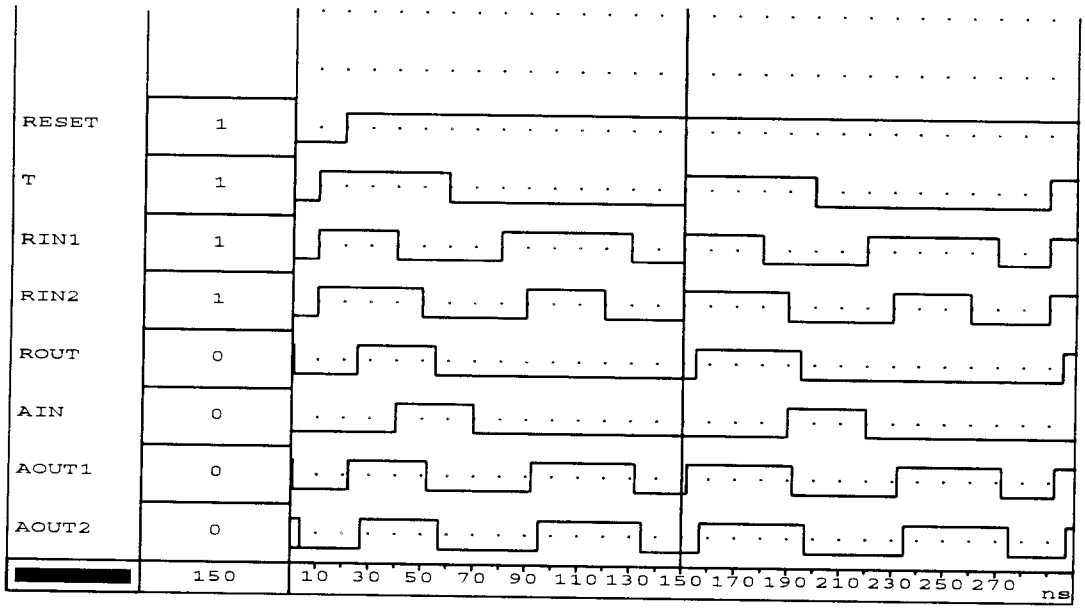

Figure 5.16 Simulation result of a 4-cycle Data Flow TRUE actor.

flip-flop is connected to the D input of the same flip-flop. The output Q will hold its current state. This can be observed in the simulation result presented in Figure 5.14

Now we turn our attention to the 4-cycle pipelined implementation of a TRUE actor, shown in Figure 5.15. The corresponding simulation result is shown in Figure 5.16. Once the two request inputs have arrived and the circuit has accounted for no continual-feeding and run-away conditions, the Boolean value of T comes into play. If the value is true, then the reader will find the schematic self-explanatory. There is the interesting feature that output acknowledge for the control token A_{out2} is derived from R_{out}. As a result, we need an additional AND gate and OR gate to generate A_{out2} when R_{out} has not been generated (because T=0).

As stated before, the FALSE actor is similar to the above implementations of the TRUE actor except for an additional inverter in the path of the controlling Boolean signal.

5.3.4 Self-timed implementations of the data flow MERGE actor

This is the most complex actor. Because of its intricate firing rule, the event control request and acknowledge signals for a 2-cycle implementation can all be out-of-sync. The schematic is presented in Figure 5.17, and the simulation result is presented in Figure 5.18. We thus have initial Muller-C elements to avoid run-away and continual-feeding conditions. There are also AND gates in the path of A_{out}_T and A_{out}_F controlled by the Boolean signal C, allowing either the token on the T input arc, or the token on the F input arc to be consumed (if both are present).

Because of the many possible combinations, we use a 2-cycle to 4-cycle converter for the request inputs. This converter is basically a double-edge triggered flip-flop that has its D input connected to logic '1'. Once the valid condition is

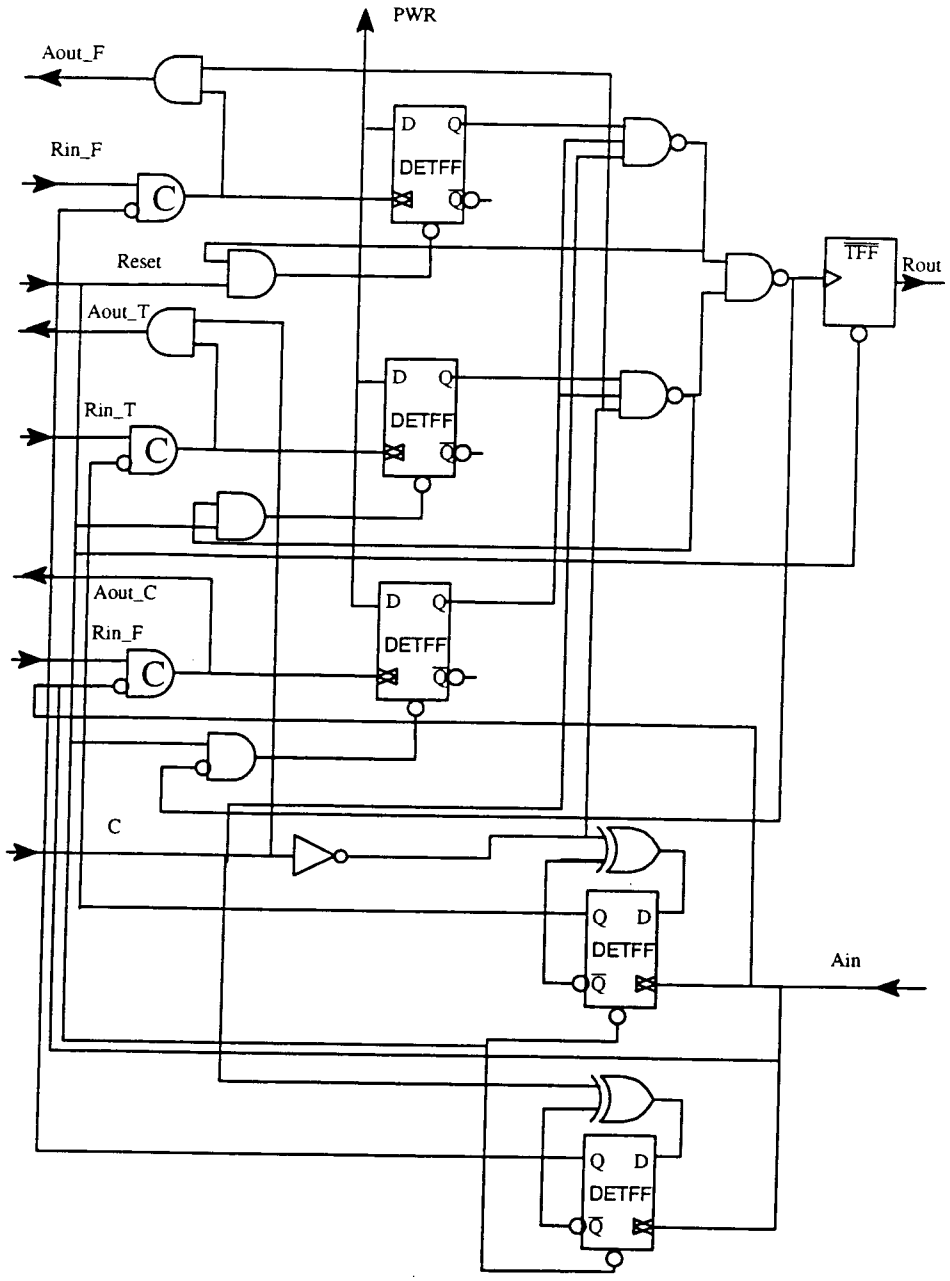

Figure 5.17 2-cycle implementation of a Data Flow MERGE actor.

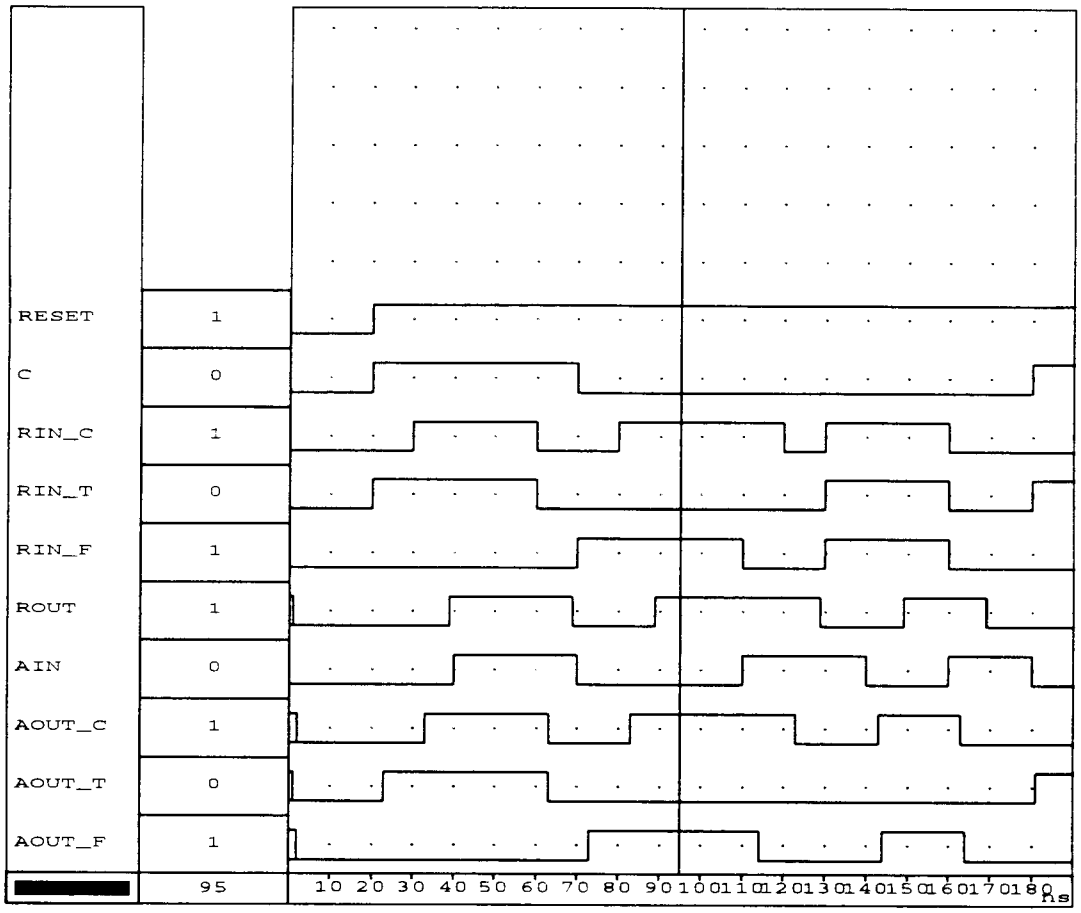

Figure 5.18 Simulation result of a 2-cycle Data Flow MERGE actor.

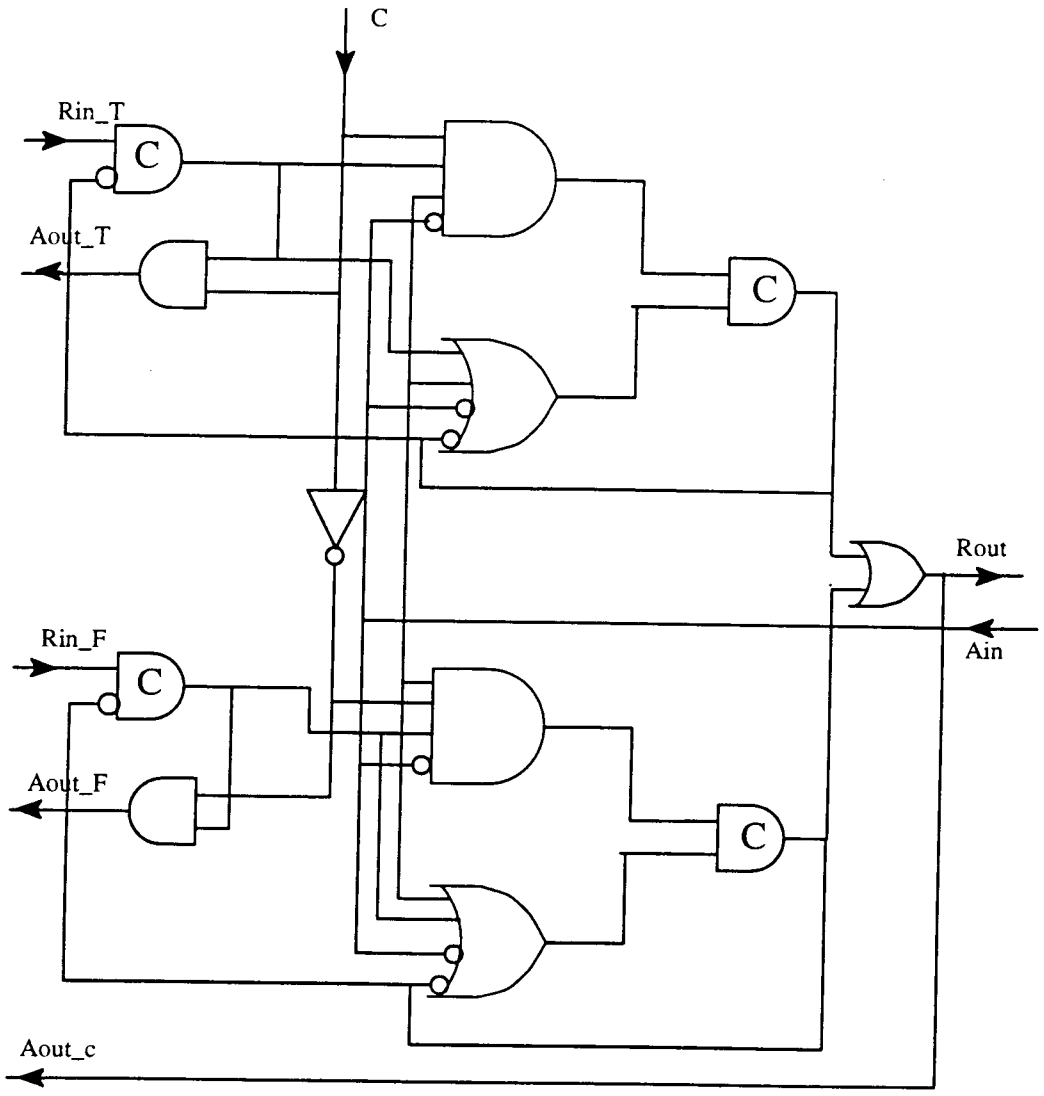

Figure 5.19 4-cycle implementation of a Data Flow MERGE actor.

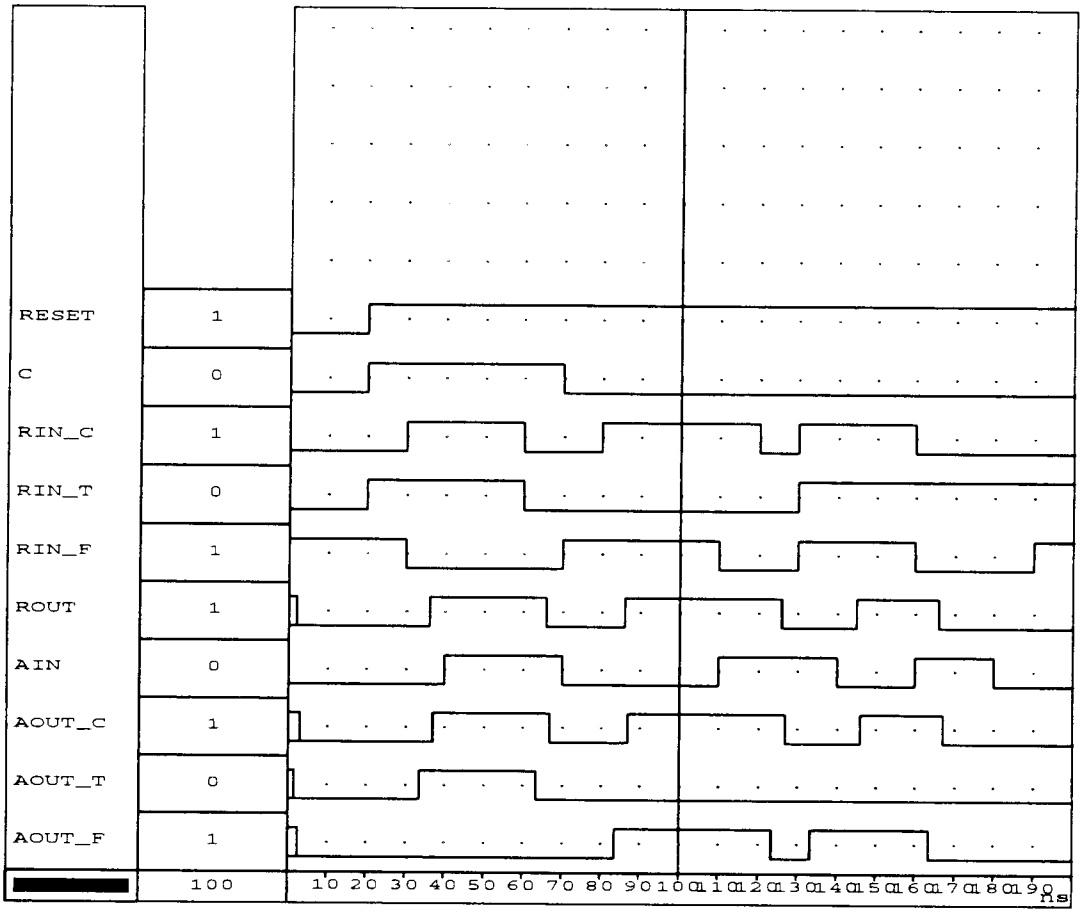

Figure 5.20 Simulation result of a 4-cycle Data Flow MERGE actor.

attained, denoted by the output of the corresponding NAND gate going high, the flip-flop is reset. At the output, a toggle flip-flop is used to act as a 4-cycle to 2-cycle converter.

The feedback path has two FF/EX-OR gate structures (discussed in the previous section), one for each of the two data input arcs. The functionality should be clear to the reader.

The 4-cycle pipelined implementation of the MERGE data flow actor is illustrated in Figure 5.19. It can be observed that the MERGE actor extends very nicely from the TRUE actor implementation in the previous section. Notice the AND gates in the paths of the acknowledge outputs. The simulation result is illustrated in Figure 5.20.

5.3.5 Self-timed implementation of the data flow SELECT actor

Unlike the 2-cycle MERGE actor, the 2-cycle SELECT actor will have its two request inputs - data and control in-sync. Thus, the 2-cycle to 4-cycle and 4-cycle to 2-cycle converters can be dispensed with. Note that the request and acknowledge inputs can become out-of sync, resulting in the need of the {FF/EX-OR gate} structure to remember the sense of the previous transition on that particular output arc. Figure 5.21 and Figure 5.22 illustrate the schematic of the 2-cycle pipelined implementation and the simulation results of the SELECT actor, respectively.

As will be seen later, the SELECT actor has the opposite functionality of the MUX actor. In fact, the 4-cycle pipelined implementation is similar to Meng's demultiplexer, as shown in Figure 5.23. The simulation result of the corresponding circuit is shown in Figure 5.24.

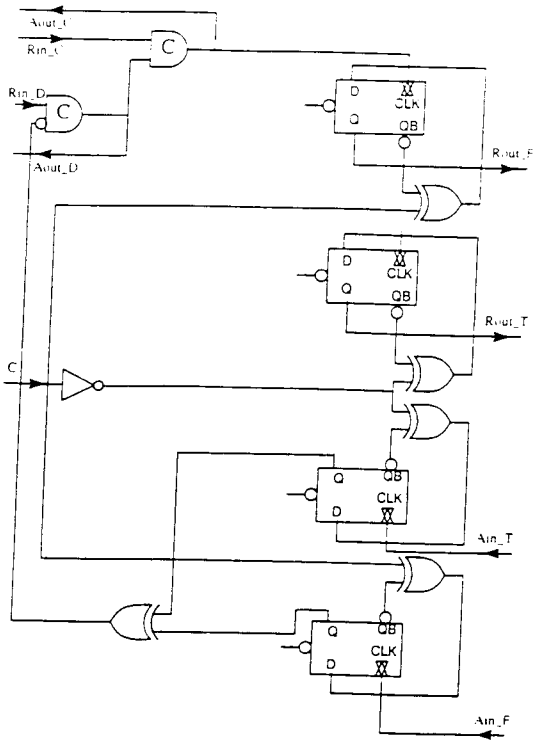

Figure 5.21 2-cycle implementation of the Data Flow SELECT actor.

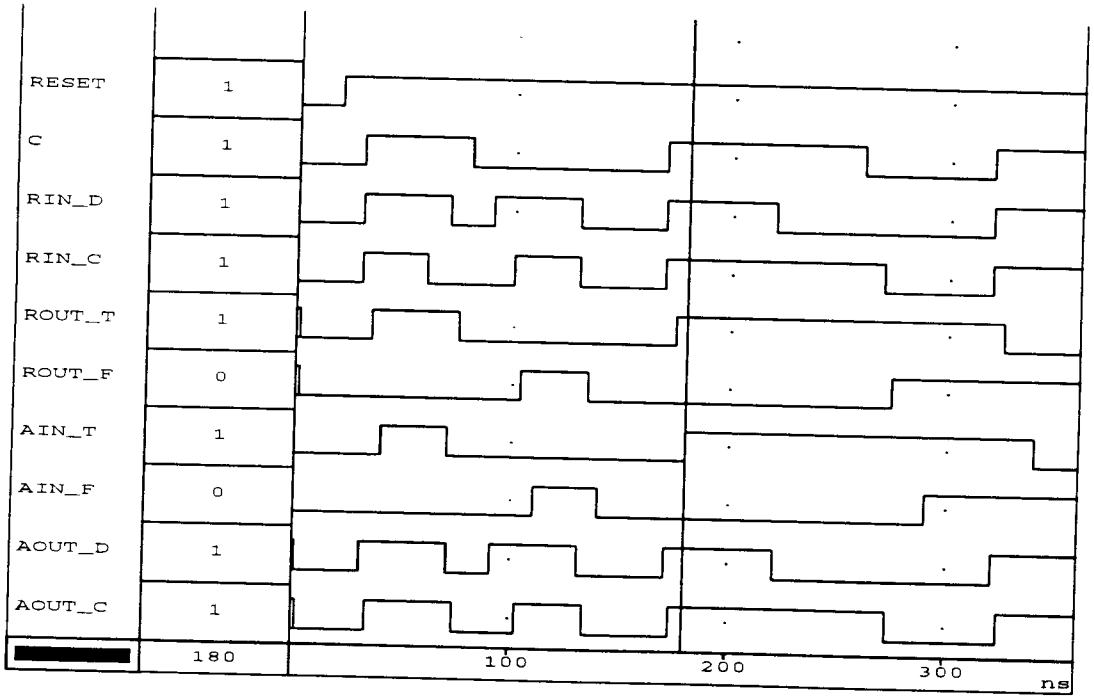

Figure 5.22 Simulation result of a 2-cycle Data Flow SELECT actor.

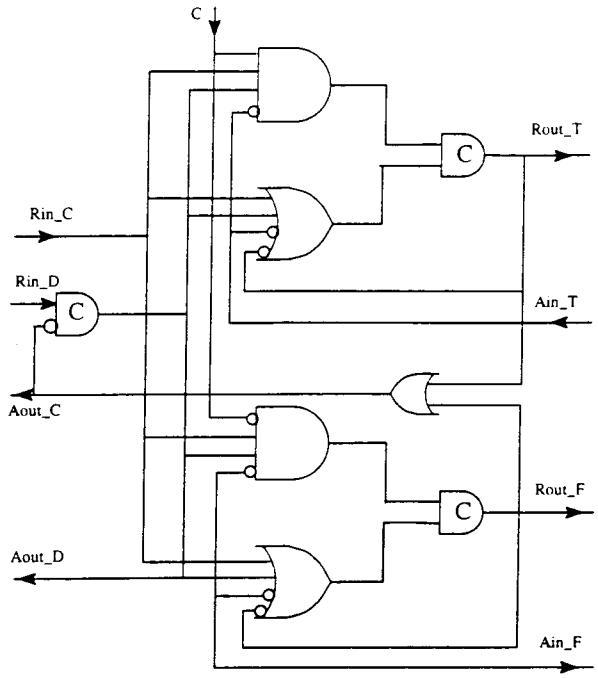

Figure. 5.23 4-cycle implementation of a Data Flow SELECT actor.

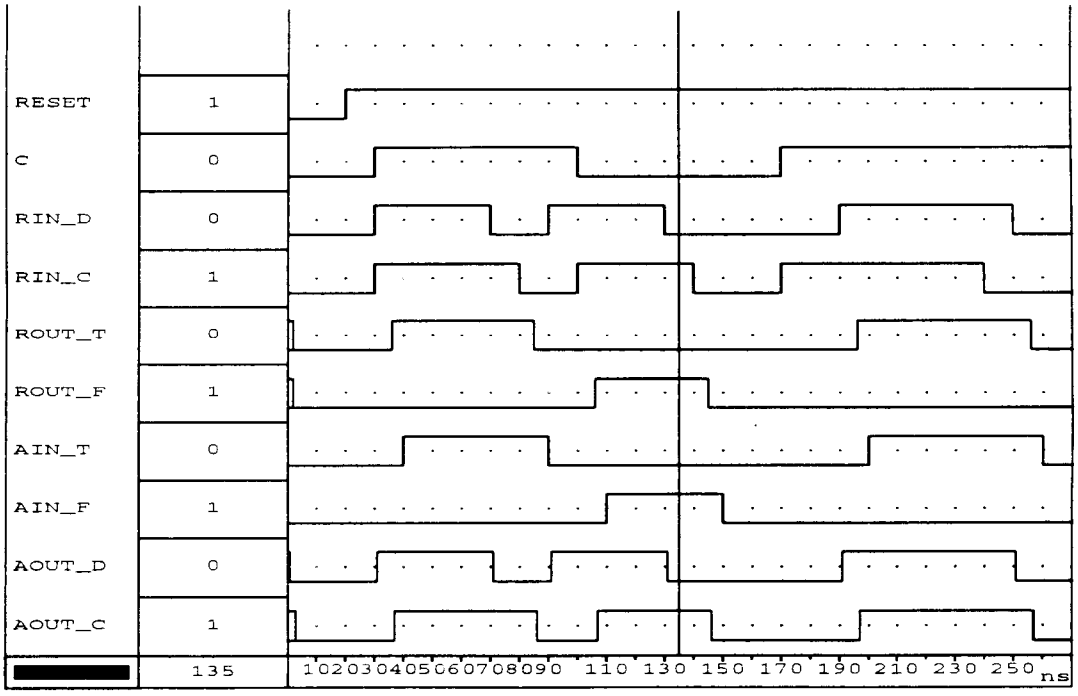

Figure 5.24 Simulation result of a 4-cycle Data Flow SELECT actor.

5.3.6 Self-timed implementation of the data flow MUX actor

From Chapter 3, we know that the MUX actor fires only when all 3 input tokens are available and no output token is present. This results in the production of an output token after the consumption of the input tokens. Thus, it can be viewed as a slightly complex PREDICATE actor. Since all the inputs (i.e. both request and acknowledge) will be in-sync, the circuit should be intuitively simple.

An important result is that the Boolean value C is not required in the design of the control path of this actor. It can be used along with the request signals to control the actual multiplexer in the data path. The 2-cycle pipelined and the 4-cycle pipelined implementations are shown in Figure 5.25 and Figure 5.27, respectively. The corresponding simulation results are shown in Figure 5.26 and Figure 5.28, respectively.

5.3.7 Self-timed implementation of the data flow INIT actor

The INIT actor is the only actor that does not have all of its signals in the same state on system reset. In fact, the Rout signal is logic '1' on system reset which, on abstracting up to the data flow graph level denotes the existence of an initial token.

As can be seen from Figure 5.29, a 2-cycle pipelined version of it can be easily implemented using a Muller-C element that gets 'set' on system reset. An inverter in the output acknowledge path guarantees that the rest of the pipeline will work correctly. The simulation result is illustrated in Figure 5.30.

A similar principle is followed in designing the 4-cycle pipelined INIT actor. An additional Muller-C element of the type discussed above is used. Figure 5.31 and Figure 5.32 illustrate the schematic and simulation result of this circuit.

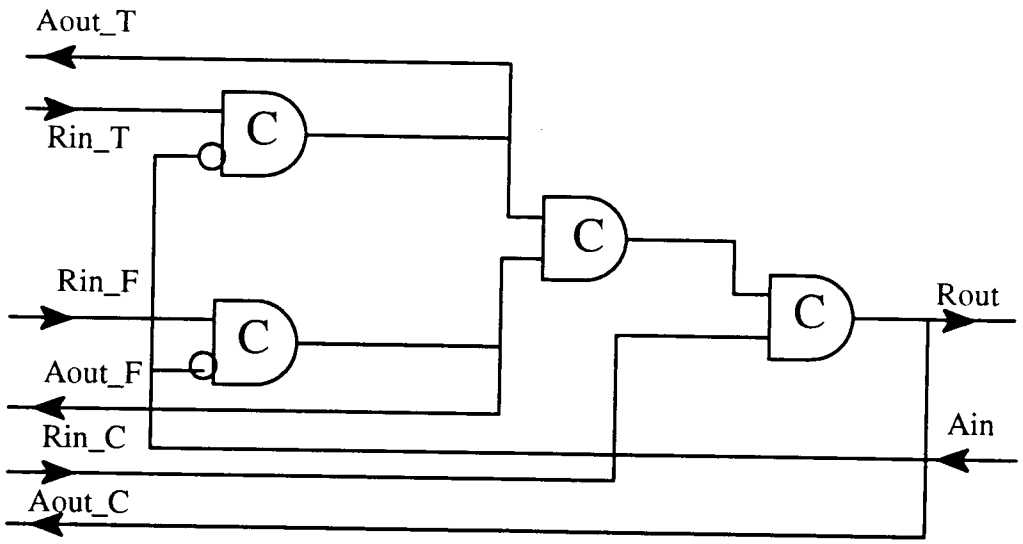

Figure 5.25 2-cycle implementation of the Data Flow MUX actor.

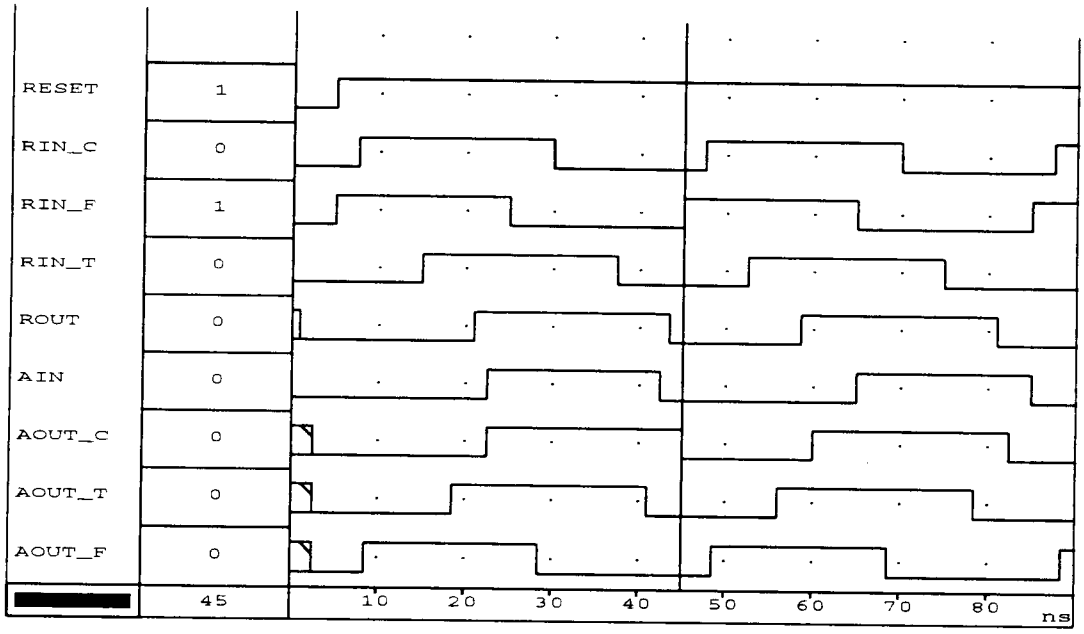

Figure 5.26 Simulation result of a 2-cycle Data Flow MUX actor.

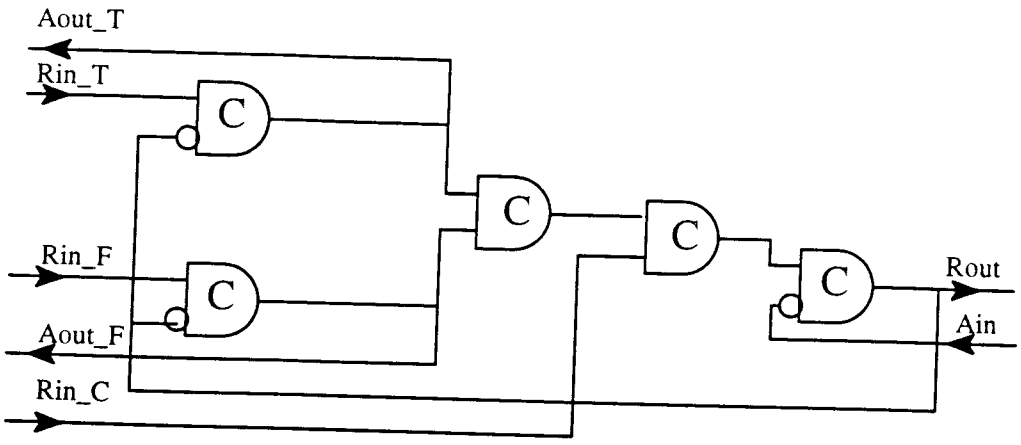

Figure. 5.27 4-cycle implementation of a Data Flow MUX actor.

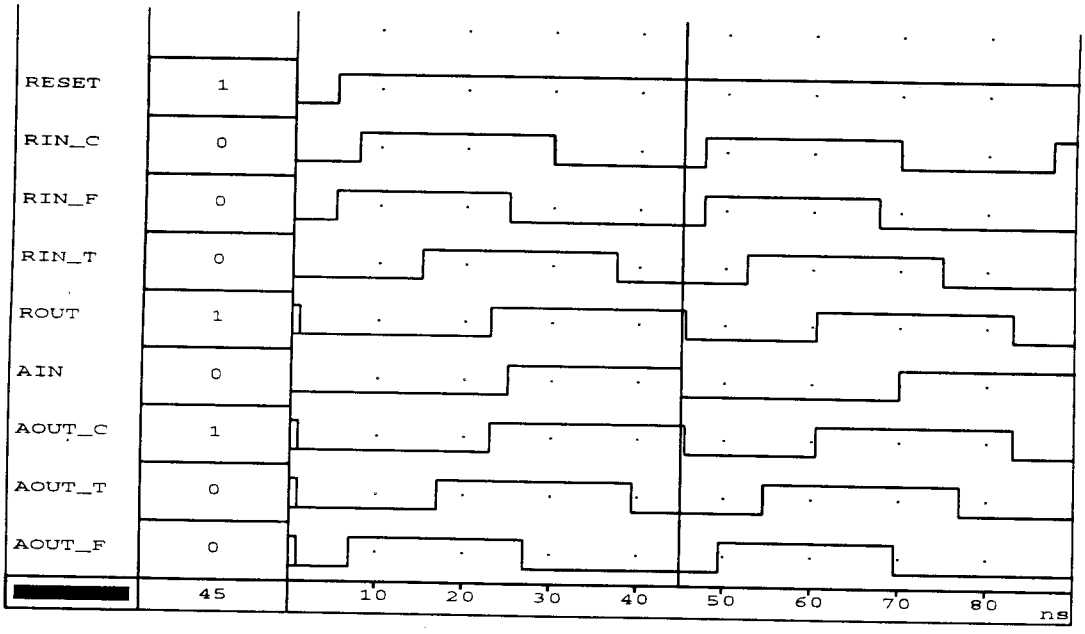

Figure 5.28 Simulation result of a 4-cycle Data Flow MUX actor.

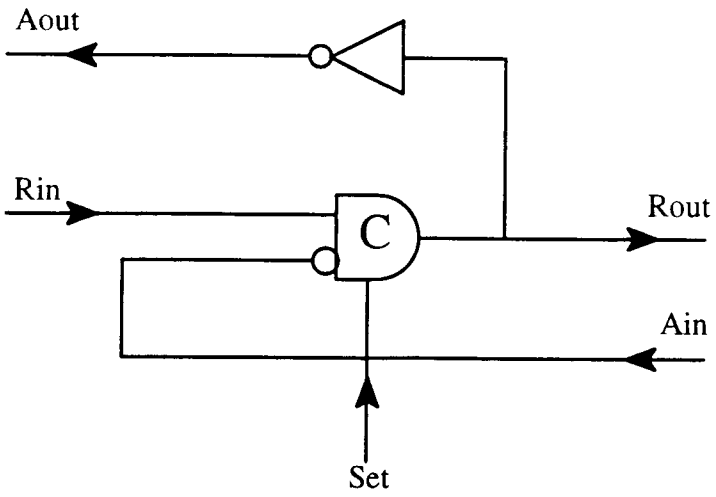

Figure 5.29 2-cycle implementation of the Data Flow INIT actor.

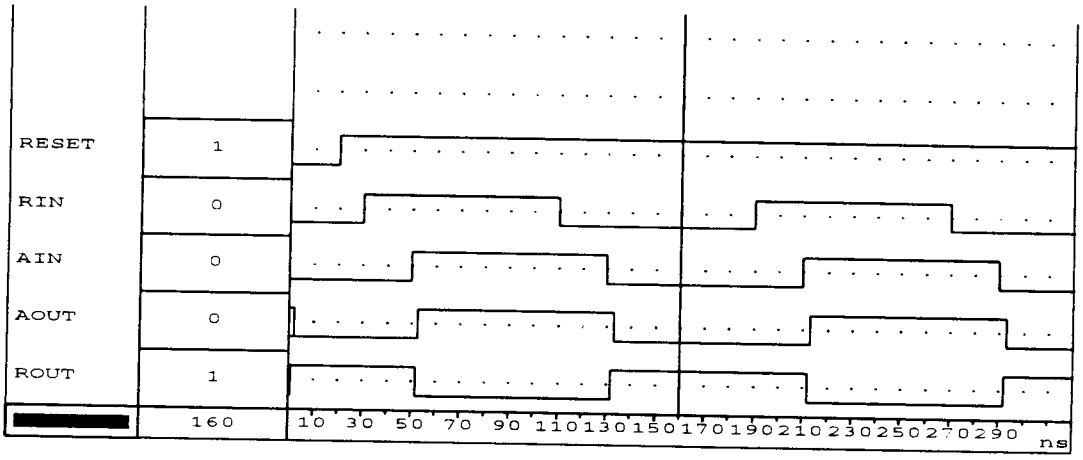

Figure 5.30 Simulation result of a 2-cycle Data Flow INIT actor.

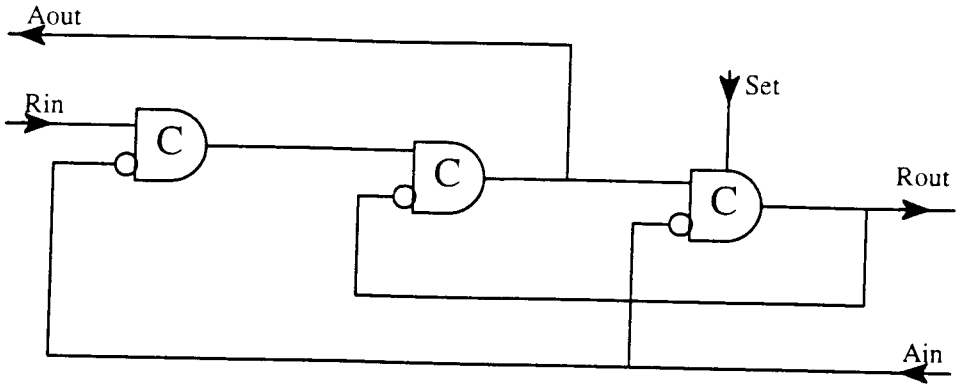

Figure. 5.31 4-cycle implementation of a Data Flow INIT actor.

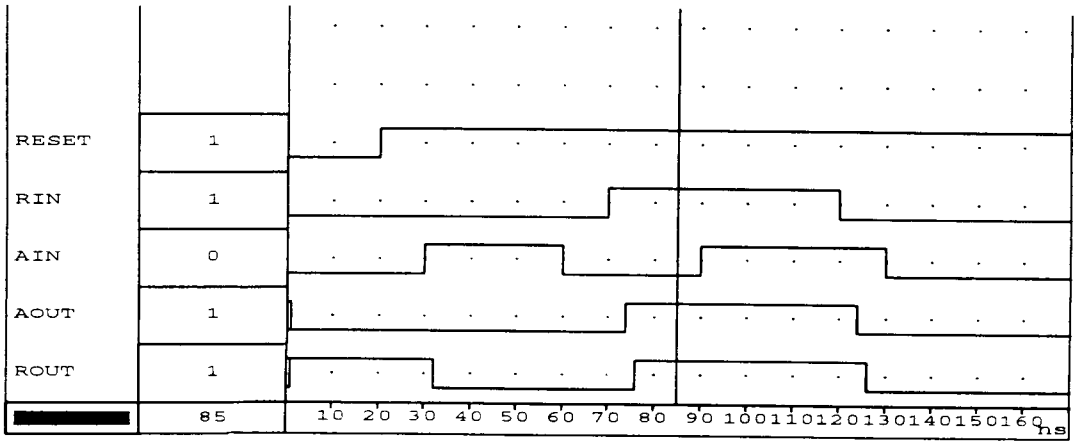

Figure 5.32 Simulation result of a 4-cycle Data Flow INIT actor.

5.3.8 Self-timed implementation of the data flow IDENTITY actor

This is a simple sample delay actor, usually used in the feedback path of DSP algorithms. Therefore, it is exactly like the basic pipeline event control circuits discussed at the beginning of this chapter.

5.3.9 Self-timed implementation of the data flow COUNTER actor

This is a very interesting actor. As discussed in Chapter 3, this actor consumes N input tokens before producing an output token. This at the circuit level requires N input request transitions in the 2-cycle protocol and 2N-1 input transitions in the 4-cycle protocol before a transition is produced on the output.

We began by using a conventional ripple COUNTER. Note that any counter design methodology can be used, since all we are interested in is the signal that indicates that the count has been reached. Of course, the same trade-offs of hardware cost versus speed that apply to counter designs are applicable here. Therefore, for the purposes of clarity in the schematics, we will denote this counter as a block with an input and an output. The extra control circuitry that is required to enable this counter to act as a COUNTER actor is also shown.

The reader should by this time be familiar with the circuit schematics. In fact, the solutions proved to be very elegant and minimal. The schematic and simulation results of the pipelined 2-cycle COUNTER actor are shown in Figure 5.33 and Figure 5.34, respectively. The 4-cycle counterparts are illustrated in Figure 5.35 and Figure 5.36, respectively.

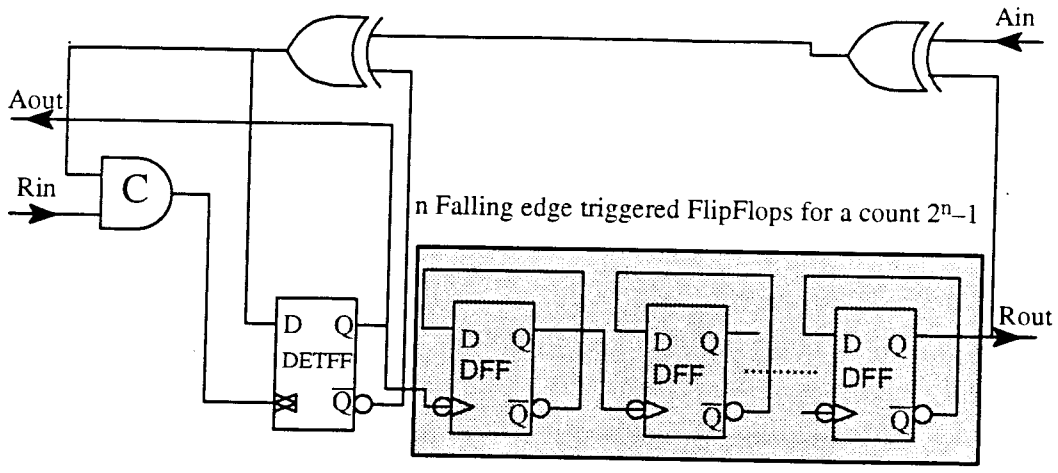

Figure 5.33 2-cycle implementation of the Data Flow COUNTER actor.

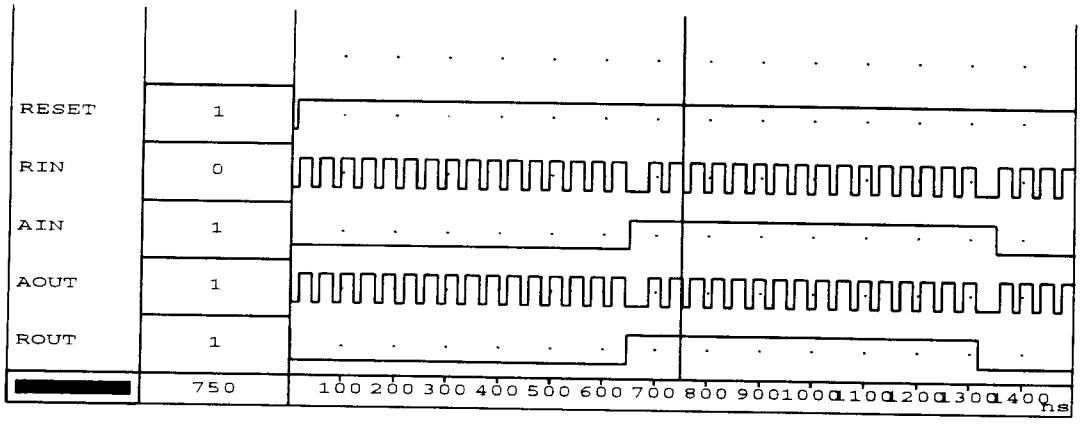

Figure 5.34 Simulation result of a 2-cycle Data Flow COUNTER actor.

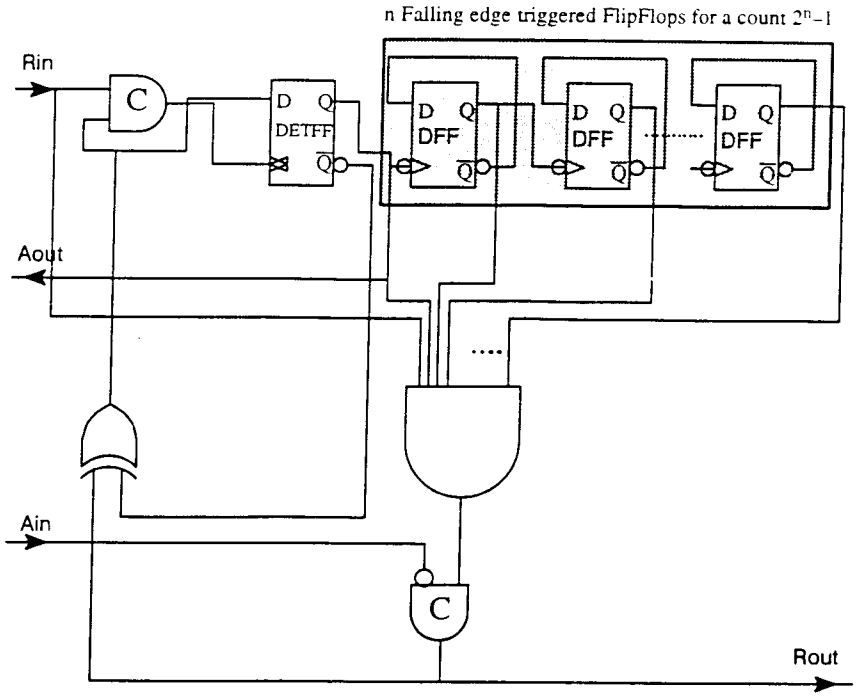

Figure. 5.35 4-cycle implementation of a Data Flow COUNTER actor.

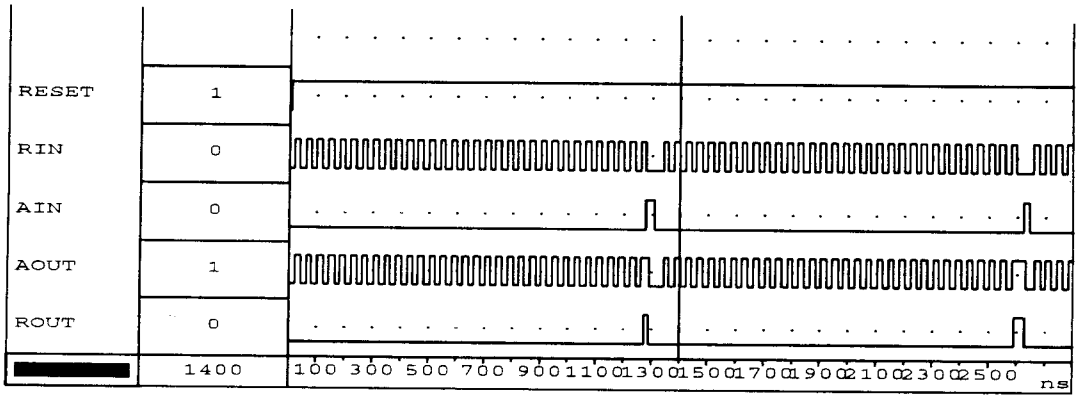

Figure 5.36 Simulation result of a 4-cycle Data Flow COUNTER actor.

5.3.10 Self-timed implementation of the data flow REPEAT actor.

This actor is in essence the exact opposite of the COUNTER actor. Here, on the consumption of one input token, N output tokens have to be created. This actor was probably the most difficult to visualize. The solution is surprisingly simple. By using the COUNTER actor that was designed in the previous section and with some additional control circuitry, we could design a REPEAT actor.

It can be observed in the schematic of the 2-cycle version that the COUNTER actor is represented by a block with the four basic request and acknowledge signals. A double-edge triggered flip-flop, which is 'set' on system reset, is used along with two 2-input AND gates and a 2-input EX-OR gate to channel the generation of the output request signal. The first output transition is generated using the input request signal. Since the input acknowledge signal of the REPEAT actor is fed into the input request signal of the COUNTER actor, the rest of the N-1 transitions are generated using the output acknowledge signal of the COUNTER actor.

The Nth output transition causes the COUNTER actor to produce a transition at its request output. This output is connected to the acknowledge input of the COUNTER actor and also to the clock input of the D flip-flop. The D input of the D flip-flop holds a logic '0', which on the arrival of the clock is passed to the Q output and then into the SET input of the same flip-flop. The circuit is now ready to proceed with the next transaction.

This might seem confusing to the reader. It is recommended to trace the operation using the schematic in Figure 5.37 and the simulation result in Figure 5.38.

The corresponding 4-cycle circuit schematic is very similar in principle and is represented in Figure 5.39. The simulation result is in Figure 5.40.

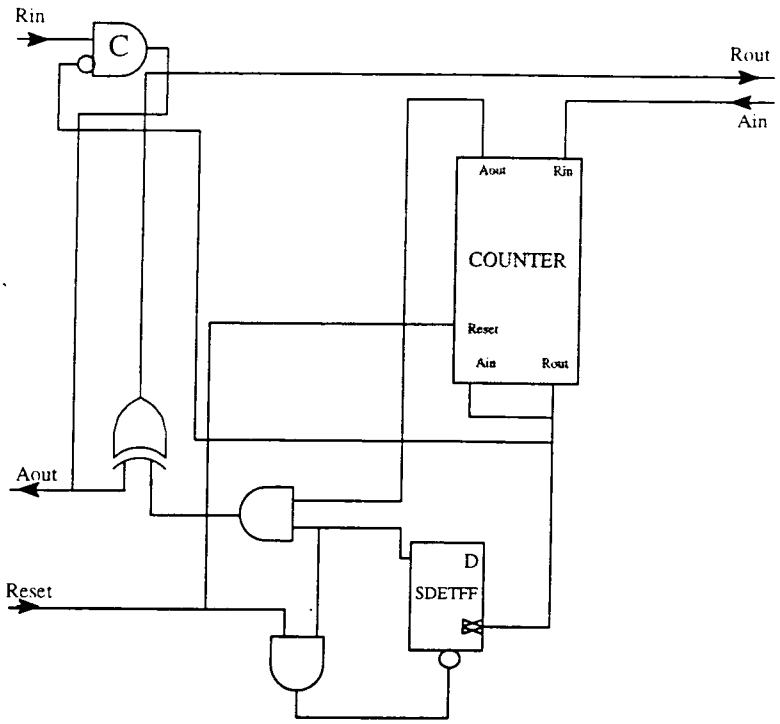

Figure 5.37 2-cycle implementation of the Data Flow REPEAT actor.

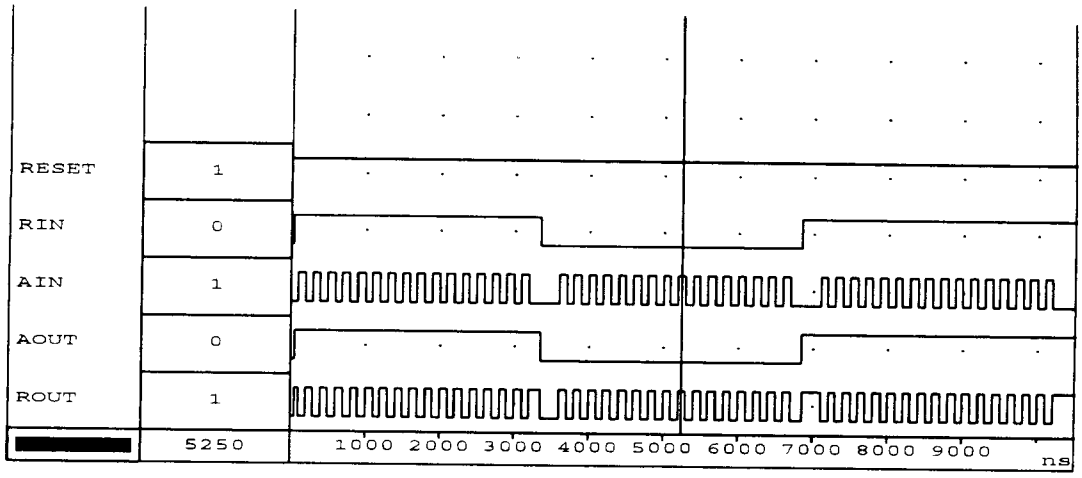

Figure 5.38 Simulation result of a 2-cycle Data Flow REPEAT actor.

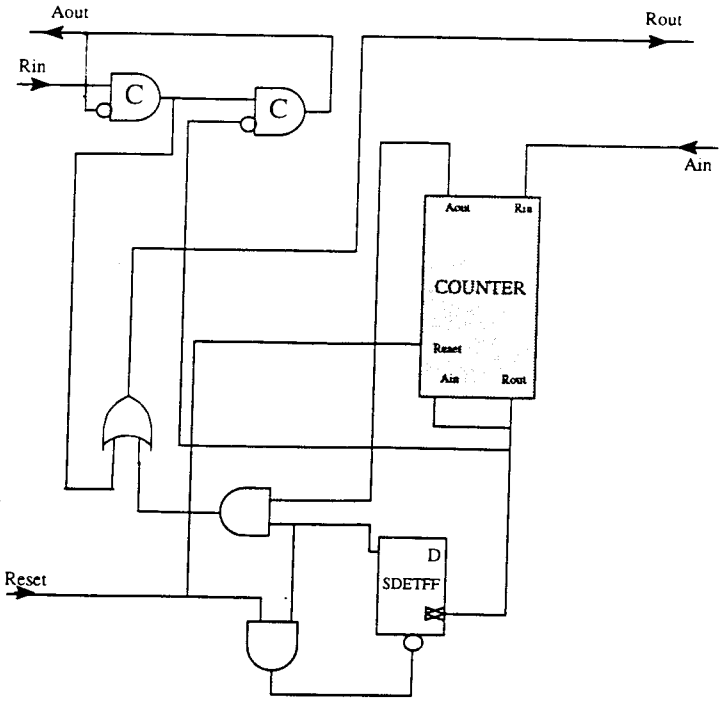

Figure. 5.39 4-cycle implementation of a Data Flow REPEAT actor.

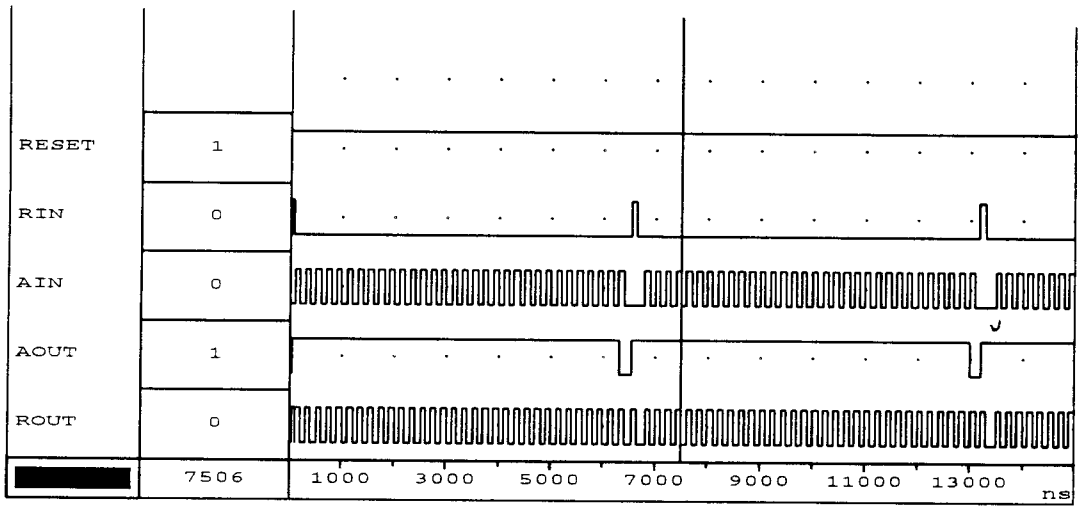

Figure 5.40 Simulation result of a 4-cycle Data Flow REPEAT actor.

Note that a shift actor can use exactly the same control. Only the data path changes complexion. Instead of an one n-bit register in the case of the REPEAT actor, the shift actor requires n 1-bit registers.

5.4 Comparison of the Two Protocols

After presenting the various implementations, the next logical step was to compare the two protocols based on the various implementations of the different data flow actors. A very simple benchmark for comparison has been used. This benchmark is based on a few assumptions.

- We assume that the circuits will be implemented using static CMOS.
- Secondly, we calculated the transistor count of all types of gates, flip-flops, and Muller C-elements used. The transistor count is representative of the silicon area that will be required by that particular primitive.
- Then we calculated the intrinsic delay of each gate in terms of inverter delays. This assumption makes the calculations independent of the technology used to implement the gates. Worst case conditions have been considered.
- Table I contains the transistor counts and the intrinsic gate delays of the various 2-input gates, a Muller-C element with reset, a Muller-C element with one inverted input and reset, a positive-edge triggered flip-flop and a double-edge triggered flip-flop. For every additional input to a logic gate, we considered an additional two transistors and half a gate delay.

Table I. Transistor count and intrinsic gate delay of the basic logic elements for a static CMOS implementation. Worst case conditions are considered. Note that every additional input to a logic gate adds 2 transistors and 0.5 gate delay.

Logic element	Transistor count	Gate Delay
Inverter	2	1
Buffer	4	2
2-input NAND gate	4	1.5
2-input AND gate	6	2.5
2-input NOR gate	4	2
2-input OR gate	6	3
2-input XOR gate	10	2.5
Muller-C element	13	2.5
Positive-edge triggered D FF	9	3
Double-edge triggered D FF	26	3

- While calculating the intrinsic delay of the 4-cycle implementation of the data flow actors, we multiplied the delay by 2. Note that the same circuit is used to make all the control signalsreturn to zero.

In Table II, we present the transistor counts and intrinsic gate delays of the various implementations of the data flow links and actors. As stated earlier, to be concise we have not presented the circuit schematics of the non-pipelined implementations. As these have been designed and simulated, we will present the

Table II. Comparison of 2-cycle and 4-cycle pipelined implementations of the data flow links and actors. Note that the transistor count, intrinsic delay and performance measure are symbolized as A, T and P, respectively.

Data Flow Actor	2-cycle protocol			4-cycle protocol			Difference (%)
	A	T	P=A*T	A	T	P=A*T	
PRIMITIVE	17	4.5	76.5	30	14	420	449
LINK	28	8	224	58	19	1102	392
PREDICATE	43	6	258	58	19	1102	327
TRUE	116	17	1972	83	32	2656	35
MERGE	258	21.5	5547	132	28	3696	-33
SELECT	188	22	4136	92	39	3588	-13
MUX	56	8.5	476	71	24	1704	258
INIT	17	4.5	76.5	45	21	945	1135
COUNTER	59	10.5	619.5	54	27	1458	135
REPEAT	122	24.5	2989	128	94	12032	303

above numbers for them also. The transistor count is symbolically represented as 'A' and the intrinsic delay of the actor as 'T'.

For an effective comparison, we calculate the area-delay product, P (= A*T). The resulting numbers are presented in the same table. This number is compared for the two protocols and presented as a percentage. A positive number means the 2-cycle implementation is better and vice versa.

Similarly, Table III contains the corresponding figures for non-pipelined implementations for both the protocols. Note that non-pipelined implementations for

Table III. Comparison of 2-cycle and 4-cycle non-pipelined implementations of the data flow links and actors. Note that the transistor count, intrinsic delay and performance measure are symbolized as A, T and P, respectively.

Data Flow Actor	2-cycle protocol			4-cycle protocol			Difference (%)
	A	T	P=A*T	A	T	P=A*T	
PRIMITIVE	8	4	32	49	24	1176	3575
LINK	21	6.5	136.5	50	29	1450	962
PREDICATE	21	6.5	136.5	67	26	1742	1176
TRUE	101	13.5	1363.5	118	40	4720	246
MERGE	201	18	3618	149	44	6556	81
SELECT	173	19	3287	97	34	3298	0.33
MUX	34	9	306	88	28	2464	705

the INIT, COUNTER and REPEAT actors were not designed. Since the INIT actor is usually in the feedback path in DSP algorithms and a self-loop should have one sample delay, it was felt that only a pipelined version was required. Another reason was that an INIT actor signified that an initial token is being held in a register (i.e. a pipelined implementation is required).

As far as the COUNTER and REPEAT actors were concerned, they were large grain actors, in which case a non-pipelined implementation would result in their latency greatly affecting the performance of the system.

Comparing the simple benchmark numbers presented in the last column of Table II and Table III, it is very easy to see that the 2-cycle protocol is by far the better alternative. Only the pipelined 2-cycle implementation of the MERGE and

SELECT actors are worse than their 4-cycle counterparts. The main cause of this is the relatively expensive double-edge triggered flip-flop that was used in the 2-cycle implementations.

5.5 Summary

The circuit schematics and the results presented in this chapter are the main contribution of this thesis. This should conclusively prove that the 2-cycle protocol offers greater potential than the 4-cycle protocol in the design of the distributed control path of a computation algorithm - application-specific or general purpose.

In the next chapter, we will investigate these conclusions further in the context of an application-specific DSP algorithm.

Chapter 6. SIMULATION OF A SELF-TIMED DECIMATION FILTER

6.0 Introduction

In the last chapter, we discussed the 2-cycle and 4-cycle self-timed implementations of the data flow actors. After a comparison of the transistor counts (related to cost in terms of silicon real-estate), and knowing that the intrinsic delays of the actors were comparable, it was time to simulate an actual example to determine which of the two protocols offered greater system level performance. Intuitively, it seemed that the 2-cycle philosophy should be able to offer at least a 100% system performance improvement. But does this intuition translate to reality? The determination of this fact is the focus of this chapter.

We needed an example that would be a good benchmark for the event-control circuits and not a commentary on the actual implementation of the computation blocks. We decided on a multistage multirate comb filter that would be used for the purpose of decimation. The example does not have any decision making, which is not true of all DSP applications. Also, the transistor counts of the 2-cycle decision actors is greater than that of the 4-cycle decision actors. So although the transistor count might seem unfairly in the favor of the 2-cycle protocol, the issue here is performance. We will discuss this issue once we have the actual numbers.

6.1 A Multistage Multirate Combs Filter Design Method

In [57], Chu and Burrus presented a new multistage multirate digital filter design method. Multirate filters are members of a class that has different sampling rates in various stages of the filtering operation. This class of filters includes

decimators, interpolators, and narrow-band low-pass filters implemented with decimation, low-pass filtering, and interpolation. One of its most famous applications is band-pass delta-sigma modulators.

If H(z) and D are the transfer function and decimation ratio respectively, then we need to design H(z) such that

$$H(z) = f(z)g(z^D).$$

By the commutative rule, the transfer function $g(z^D)$ can be implemented at the lower rate (after decimation) as g(z). This implementation reduces the filter order, storage requirement, and the arithmetic. Further requirements on H(z) allow only simple integer coefficients. This is feasible because there are no passband specifications on the frequency response.

A comb filter of length D is an FIR filter with all D coefficients equal to one:

$$H(z) = f(z)g(z^D) = \sum_{n=0}^{D-1} Z^{-n} = \frac{1 - z^{-D}}{1 - z^{-1}}$$

A single comb filter generally does not give enough stopband attenuation. However, cascaded comb filters can often meet the requirements. Cascading M length-D comb filters will give us the following transfer function:

$$H(z) = \left[\frac{1 - z^{-D}}{1 - z^{-1}} \right]^M$$

Thus a comb decimator has M length-D comb filters in cascade, where all the accumulators are cascaded before the sampler and all the $(1 - z^{-1})$ sections are

cascaded after the sampler. The comb filter structure has a recursive stage with a pole at z=1, which is not asymptotically stable, and therefore may overflow. The filter operation depends on a "wrap around" number system similar to the 2's-complement number system.

In our case, M=3 and D=32 and the data bus width is 20 bits. Fig. 6.1 has the C program that simulates the mathematical operations of the decimator. This program was provided to us by Richard Schreier of OSU [58]. The corresponding data flow graph is shown in Fig. 6.2.

6.2 A Self-timed Implementation of the Decimation Filter

The data flow graph of the decimation filter of Fig. 6.2 is mapped to the block diagram schematic of a self-timed implementation of it, as shown in Fig. 6.3. The reader should notice how the arcs of the data flow graph map very elegantly to the request-acknowledge control path and implicitly to the data path of the self-timed implementation.

The computation blocks have been modeled behaviorally since their design does not affect the performance of the event-control logic. The behavioral model of the 2-cycle adder is shown in Fig. 6.4. Note the modeling of the storage registers in the VHDL program. Fig. 6.5 illustrates the schematic of the 2-cycle ADDER PREDICATE. Remember that the ADDER PREDICATE has been modeled structurally. In fact, it is possible to observe the similarity of the structure to the micropipeline discussed in Chapter 2 and the 2-cycle PREDICATE actor discussed in Chapter 5. Also notice that the register is modeled as a latch-buffer combination. This is based on the "pass" and "capture" philosophy of the micropipeline.

```
/*
*       decimate.c
*       Simulates the mathematical operations carried out by a sinc^3 decimator
*       with a decimation factor of 32. The decimator uses modulo 2^20 arithmetic
*       to do its calculations.
*
*       R. Schreier of OSU, 1993.
*
*/

/**
***     Slight modifications have been made by us to reflect the functionality that
***     is shown in the data flow graph.
***
**/

#define R 32
#define N 20

static int Modulus = 1<<N;
static int Signbit = 1<<(N-1);
static int Mask = (1<<N) - 1;

int mod( int x ){
        return x&Mask;
        }

main()
{
/* Declarations */
int a1=0, a2=0, a3=0, s1=0, s2=0, s3=0;
int t, x, y1, y2, y3;

/* Get the input and go through the equations. */

for (t=o; scanf("%d",&x) > 0 ; )
{
        a1 = mod( a1 + x );
        a2 = mod( a2 + a1 );
        a3 = mod( a3 + a2 );

        if ( ++t == 32)
        {
                t = 0;
                y1 = mod( a4 - s1 );
                s1 = a4;
                y2 = mod( y1 - s2 );
                s2 = y1;
                y3 = mod( y2 - s3 );
                s3 = y2;
        }
}
exit(0);
}
```

Figure 6.1 A C program which simulates the operation of the decimator. Courtesy: R. Schreier of OSU. The original program has been modified by us.

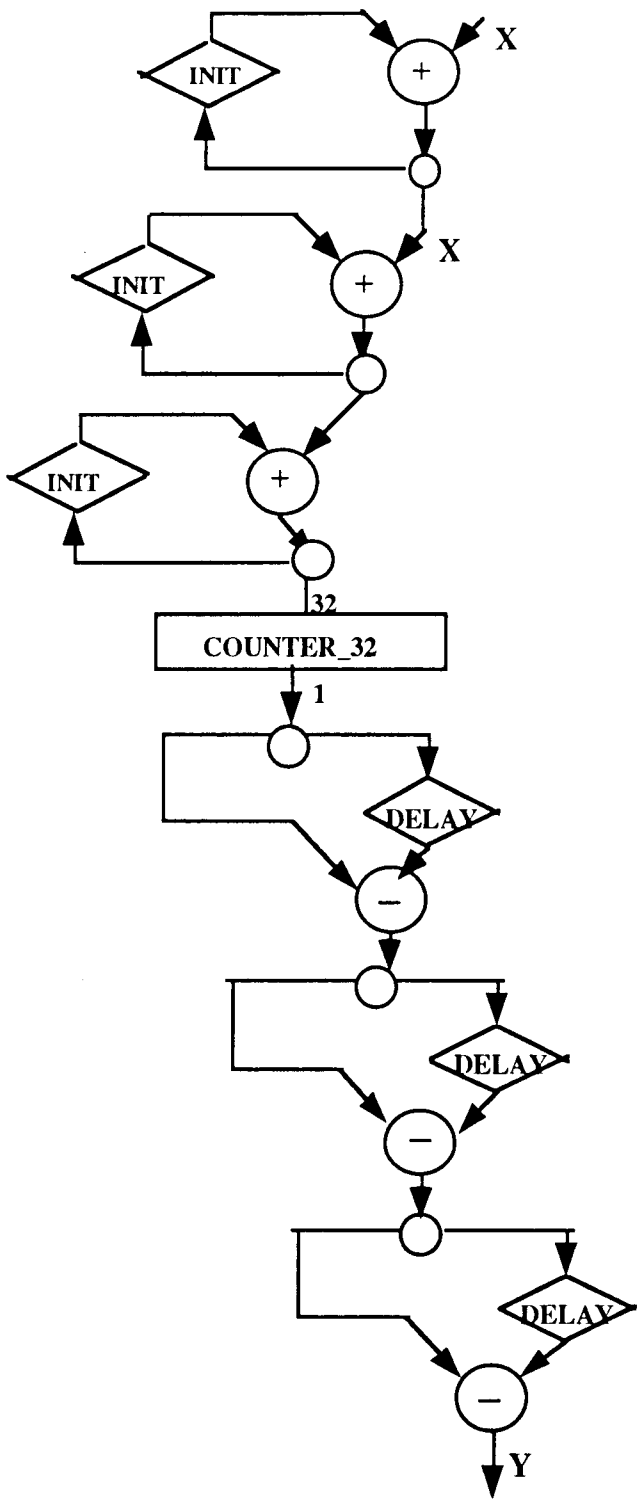

Figure 6.2 A Data Flow Graph representation of the C program in Fig. 6.1.

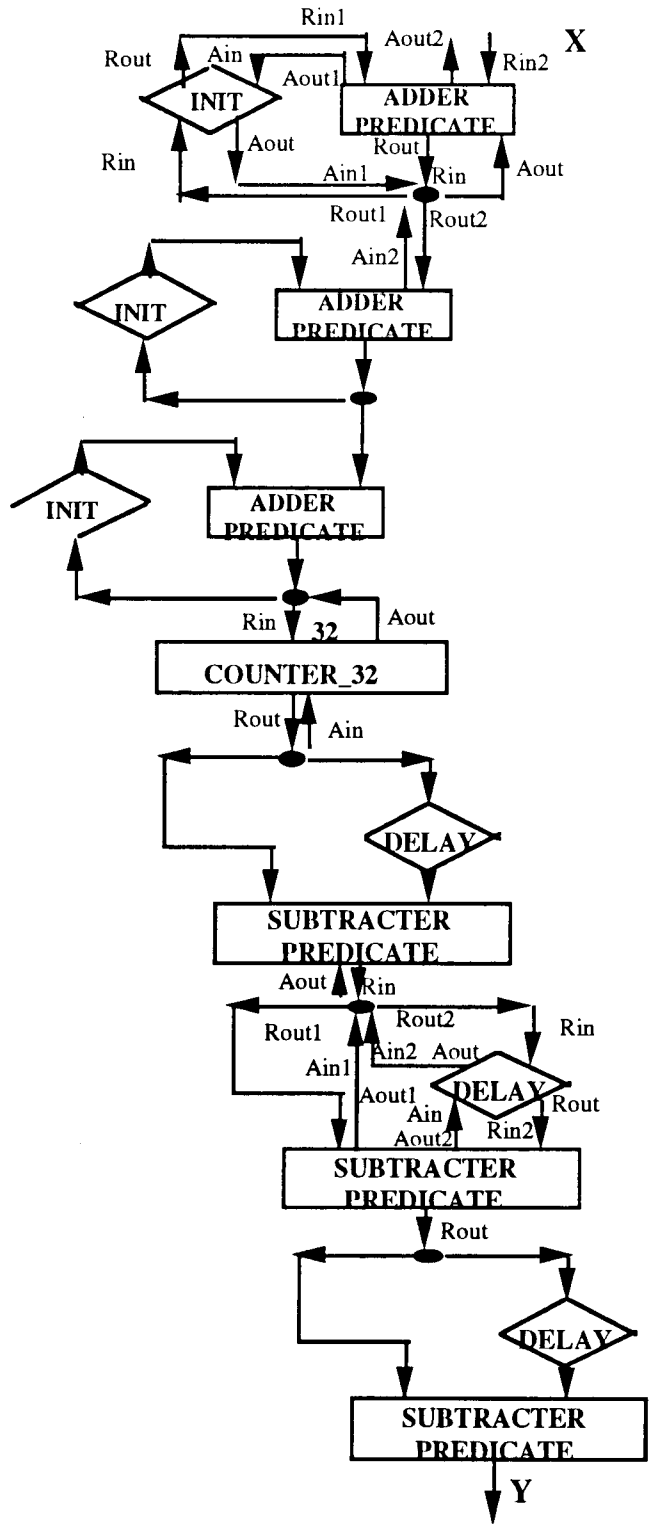

Figure 6.3 A block diagram schematic of the Data Flow program graph.

```
-- VHDL behavioral model for a 2-cycle 20-bit parallel adder with a register.
entity ADDER_20_2 is
      generic (cout_delay: time := 3500 ps; reg_delay: time := 2 ns;
                  done_delay: time := 5500 ps);
      port (signal dout: out vlbit_vector(0 to 19); signal cout: out vlbit;
            signal done: out vlbit; signal a, b: in vlbit_vector(0 to 19);
            signal cin: in vlbit; signal clk: in vlbit; signal RESET: in vlbit);
end ADDER_20_2;

architecture BEHAVIOR of ADDER_20_2 is
      signal ADDER_20out: vlbit_vector(0 to 20); -- ADDER output: cout & dout
      signal ResReg: vlbit_vector(0 to 19);      -- Result Register
      signal over: vlbit;

      begin
            addprocess: process(a, b, cin)
      variable res_22: vlbit_1d(-1 to 20); -- 22 bit temporary result
            begin
                  res_22 := add2c (add2c (a,b), '0' & cin);
                  ADDER_20out <= res_22(0 to 20);
            end process;

-- Concurrent register process statement:
-- Load up the register on both the rising and falling edges of the clock.

            register_process: process
constant XOUT: vlbit_1d(0 to 19) :=
('X','X','X','X','X','X','X','X','X','X','X','X','X','X','X','X','X','X','X','X');
constant ZERO: vlbit_1d(0 to 19) :=
('0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0');
            begin
                  if RESET = '0' then
                        ResReg <= ZERO;
                        over <= '0';
                  end if;
                  wait until pchanging(RESET) or pchanging(clk);
                        if RESET = '0' then
                              ResReg <= ZERO; over <= '0';
                        else if RESET = '1' then
                              if pchanging(clk) and over = '0' then
                                    ResReg <= ADDER_20out(1 to 20);
                                    over <= '1';
                                    else if pchanging(clk) and over = '1'
                                    then ResReg <= ADDER_20out(1 to 20;
                                          over <= '0';
                                    end if;
                              end if;
                              else ResReg <= XOUT;
                        end if;
                        end if;
            end process;

-- Concurrent signal assignments
cout <= ADDER_20out(0) after cout_delay; dout <= ResReg after cout_delay; done <=
over after cout_delay;

end BEHAVIOR;
```

Figure 6.4 Behavioral description of a 2-cycle ADDER

Intuitively, the corresponding program for the 4-cycle implementation of an adder is also the same, except that the register loads only on the rising edge. Similarly, for the 4-cycle ADDER PREDICATE the structural definition is much like the structural definition of the 2-cycle adder PREDICATE, with only one difference - the Muller-C elements used to latch data into the registers are replaced by the basic 4-cycle handshake structure shown in Fig. 2.4.

This step-by-step natural and elegant mapping of the links and actors of a data flow graph into the self-timed digital circuits of these same links and actors, resulting in a very simple pipelined structure, is the main strength of this philosophy.

6.3 Simulation Results

The final programs for the decimation filter were also written in a hierarchical fashion. In Fig. 6.3, we can see the existence of three basic modules, which act as building blocks for the whole graph. The three modules are:

Module 1 - consists of the adder PREDICATE, the LINK actor and the INIT actor

Module 2 - consists of the COUNTER actor with N = 32, and

Module 3 - consists of the subtracter PREDICATE, the LINK actor and the DELAY(IDENTITY) actor.

These modules were simulated separately and then integrated to implement the full graph. The simulation results of both the 2-cycle and 4-cycle self-timed implementations of a 20-bit 3-stage length-32 decimation filter are presented in Figures 6.6 and 6.7, respectively.

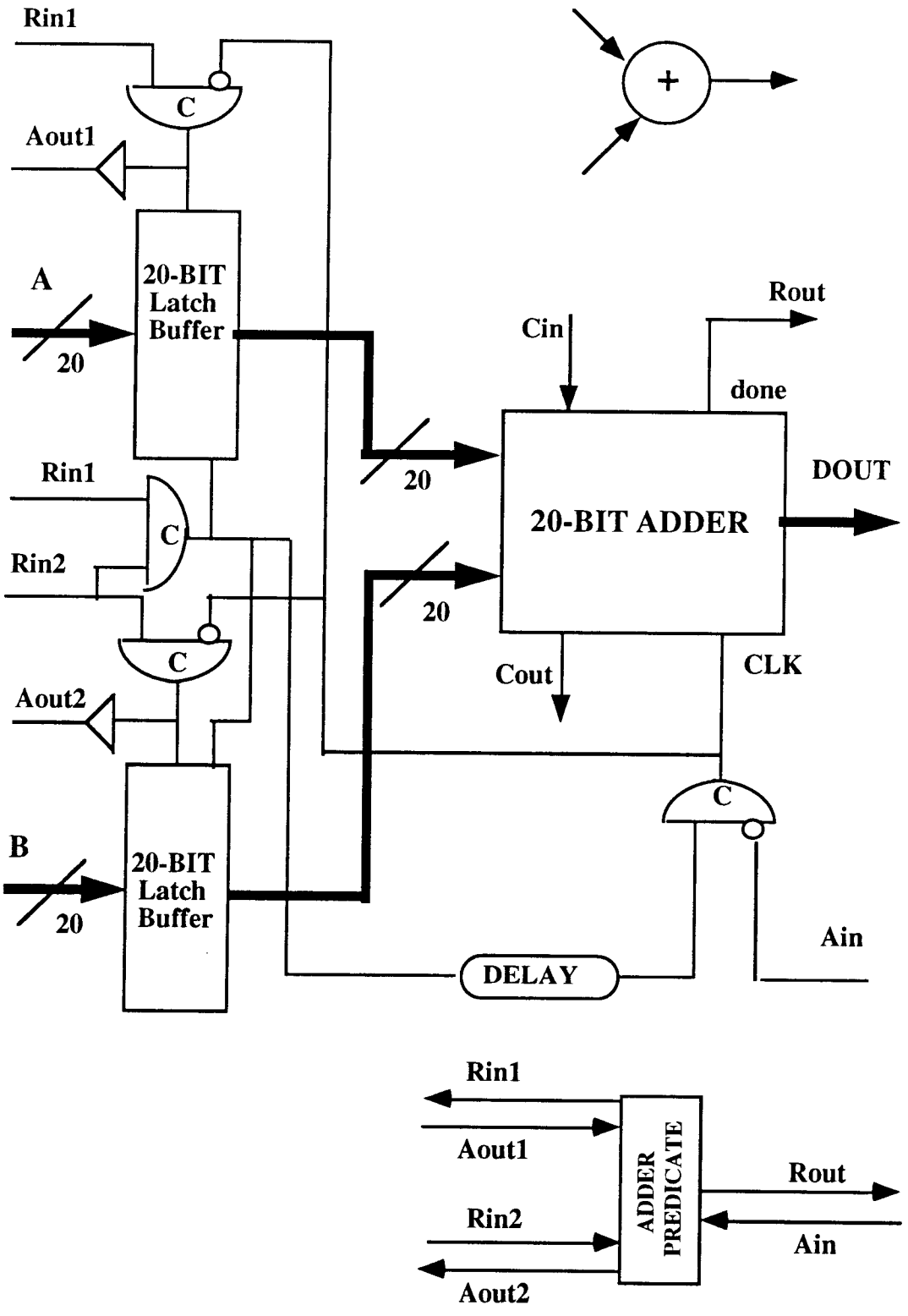

Figure 6.5 A Schematic of the 2-cycle ADDER PREDICATE.

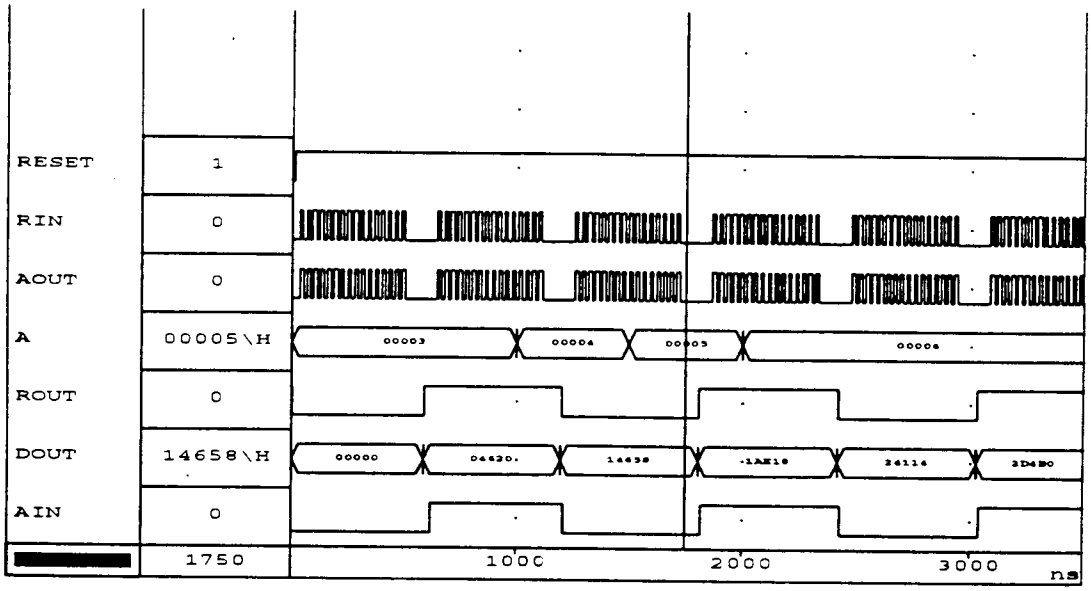

Figure 6.6 Simulation results of the 2-cycle Decimation Filter.

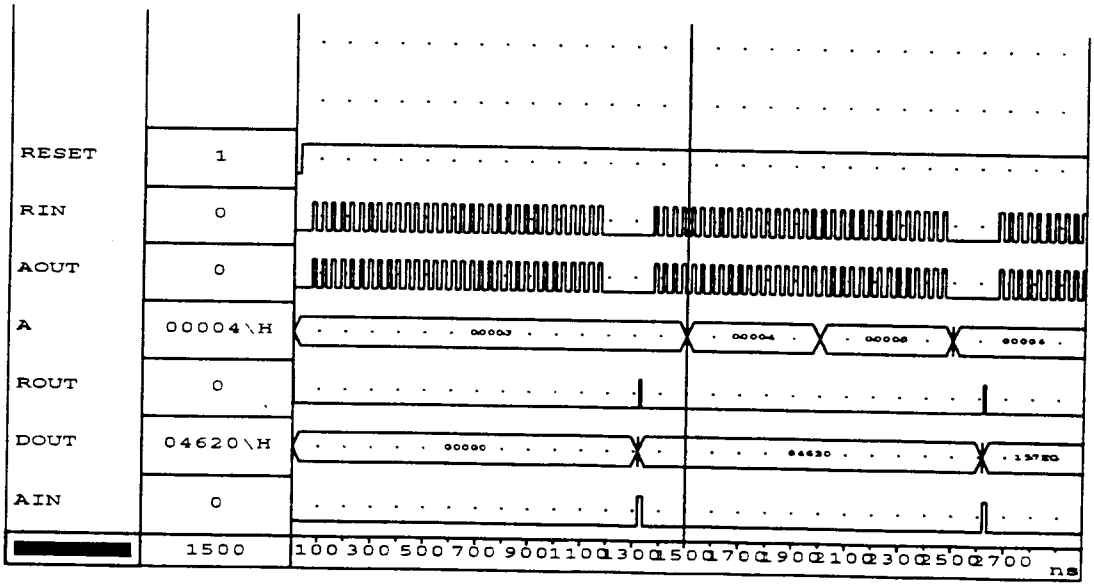

Figure 6.7 Simulation results of the 4-cycle Decimation Filter.

Comparing the results, it can be computed that the 2-cycle implementation performed better than the 4-cycle implementation by a factor of 2. Intuition translates to reality. We have answered the question we posed to ourselves at the beginning of this chapter. Also the approximate transistor count was much less (the values were computed but have not been presented). Although the decimation filter was a good benchmark to compare performance, it was not a good basis for a cost estimate. This is largely due to the lack of any decision blocks. The 2-cycle decision blocks are more expensive than their 4-cycle counterparts.

6.4 Summary

The results of this chapter allow us to safely conclude that the 2-cycle protocol offers greater performance than the 4-cycle request-acknowledge protocol. The cost might be greater hardware cost. This is dependent on the algorithm. But for most DSP applications the cost would not be considerably larger for a 2-cycle implementation than a 4-cycle implementation, if larger at all. So the cost-to-performance ratio works out be in the favor of the 2-cycle protocol.

Another important factor worth consideration is power consumption. A 2-cycle protocol is inherently more energy efficient than a corresponding 4-cycle implementation. In the next chapter, we will discuss the importance of this in greater depth.

Chapter 7. CONCLUSIONS AND FUTURE WORK

7.0 Conclusions

This thesis was an attempt to understand the issues involved in the mapping of data flow graphs adapted for DSP onto self-timed digital circuit design of the required event-control modules to generate a very elegant and natural pipeline structure.

We started by understanding the advantages of self-timed design and how, by using very simple pipeline structures we could exploit a greater degree of parallelism. The 2-cycle and 4-cycle request-acknowledge protocols were explained and compared. The use of data flow graphs adapted for DSP applications as a high level specification was discussed. The natural ease with which a data flow graph lends to the specification of self-timed circuits was demonstrated.

After designing self-timed circuits for the data flow links and actors to perform event control of the resulting pipeline, we presented an implementation of a 20-bit, 3-stage, length-32 decimation filter. The results strengthened our initial intuition that the 2-cycle protocol provided a very good way of designing self-timed digital circuits, especially in the above context.

7.1 Future Work

As with research of new ideas, there exists a plethora of problems that need to be studied, to gain a better understanding of the many issues involved. I will attempt to break these problems down into a few categories. None of these problems are trivial, but all are not equally important. By this we mean that a certain degree of

prioritizing needs to be enforced, so that the research can progress with focus and speed.

7.1.1 DSP applications

An issue of importance is to look at a larger set of DSP applications to determine what kind of graphs arise. This could help in understanding in greater depth the issues concerning automatic scheduling of data flow actors.

An example worth studying would be a common Finite Impulse Response (FIR) filter. An FIR filter computes the inner product of a vector of coefficients and a vector with the last N input tokens, where N is the order of the filter. It is assumed that it repeats forever. It fires each time a new token arrives.

Many different implementations are possible. We can have large grain approaches where the details are hidden within one actor, or a fine grain approach with multiple adders and multipliers (dependent on the order of the filter). Yet another approach is to use iteration and a single adder and multiplier.

Also we need to determine what class of algorithms (DSP or otherwise, for example floating point computations) could exploit the advantages of self-timed design. Another example is the use of self-timed circuit design principles for implementing high-rate linear adaptive filters. A vectorized version of the same will be able to handle arbitrarily high-rates, for the same hardware speed, at the cost of parallel hardware. Meng has devoted a lot of effort in studying this for a 4-cycle implementation. It would be interesting to see how a 2-cycle implementation would perform against it.

This would entail some transistor level implementations, and the design, layout, fabrication and testing of some chips/chip sets. Use of Enable/Disable CMOS

Differential Logic (ECDL) for the design of computation blocks could be an active area of study.

7.1.2 General-purpose computing

As stated above, a study of the characteristics of data flow graphs could help in understanding the complex issues involved in adapting this philosophy for general-purpose computing applications. Of course, this involves a lot of research at an architectural and systems level. Immediate applications could be floating point computation.

A very promising area of application is the design of RISC microprocessors for portable computers whose main selling point is not performance but power consumption. In [59], Dick Pountain talks about Steve Furber's research in this highly promising application. He uses the 2-cycle protocol of Sutherland.

In [60], Chang discusses the issues involved in designing a data flow microprocessor using the 2-cycle protocol. Here, he also looks at the issue of using many such processors in a multiprocessor environment.

7.1.3 CAD tool design

Finally, as in all research efforts, some customized CAD tools are required. An immediate requirement would be the development of a high-level data flow simulator to use the expanded basic data flow set as a toolkit. This will permit the designer to determine the existence of any deadlocks in the graph. It will allow him to have an initial estimate of the performance. It will allow him architecturally to revise

the graph to improve performance. Once approved, the graph could then be mapped to an automatic generation of VHDL behavioral/structural code.

Summarizing, we would like to automate the design process in the future. Design tools will be developed. These tools can be classified into four categories:

I User Interface Tools
* Schematic entry tool: A schematic capture tool for graphic entry of data flow specification (structural description).
* Compiler: A tool to convert high-level data flow description (behavior) in to structure description.

II Simulation Tools
* Simulator: A structure (logic) level simulator for data flow graph.
* Timing simulator: A tool to simulate DFG with timing.

III Analysis Tools
* Performance estimator: A static tool to predict worst case performance.
* Cost estimator: A static tool to predict and estimate the area and power for a DFG.

IV Implementation Tools
* Place and Route: Modification for DFG mapping to micropipelines.
* Logic synthesis: Auto-synthesis tool converting logic equation to ECDL.

With these tools, we will be in a position to address some important issues that have not been investigated. For example, we can then compose more complex examples for performance evaluation purposes.

REFERENCES

[1] S-L. L. Lu, "Implementation and Synthesis of Micropipelines in CMOS Differential Logic," a proposal submitted to the National Science Foundation, 1992.

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

[3] C. L. Seitz, " System Timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Eds., Addison-Wesley, 1980.

[4] C. Mead and L. Conway, *Introduction to VLSI Systems*, chapter 7, written by C. Seitz, Addison-Wesley, 1980.

[5] R. Sridhar, "Asynchronous Design Techniques," The Fifth Annual IEEE ASIC Conference and Exhibit, ASIC '92, Rochester, New York, 1992.

[6] G. Gopalakrishnan and P. Jain, "Some Recent Asynchronous System Design Methodologies," Tech. Rep. UU-CS-TR-90-016, Department of Computer Science, University of Utah, Salt Lake City, Oct. 1990.

[7] T. H. Meng, *Synchronization Design for Digital Systems*, Kluwer Academic Publishers, 1991.

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

[9] D. E. Muller and W.S. Bartky, "A Theory of Asynchronous Circuits," Proceedings of an International Symposium on the Theory of Switching, the Annals of the Computation Laboratory of Harvard University 29, Part I, Harvard University Press. Cambridge. MA, 1959.

[10] J. B. Dennis and G. R. Gao, "An efficient pipelined data flow processor architecture," Proceedings of the ACM SIGARCH Conference on Supercomputing, Florida, Nov. 1988.

[11] G. J. Myers, *Advances in Computer Architecture*, Second Edition, John Wiley and Sons, 1982.

[12] K. Hwang and F. Briggs, *Computer Architecture and Parallel Processing*, Mc-Graw Hill, 1989.

[13] J. B. Dennis, " Data Flow Supercomputers," Computer, Vol. 13, (11), pp. 48-56, Nov. 1980.

[14] Gul Agha, *ACTORS*, The MIT Press, Cambridge, MA, 1986.

[15] A. L. Davis and R. M. Keller, "Data Flow Program Graphs," Computer, Vol. 15, (2), pp. 26-41, Feb. 1982.

[16] E. A. Lee and D. G. Messerchmitt, "Static scheduling of synchronous data flow graphs for digital signal processing," IEEE Transactions on Computers, January 1987.

[17] E. A. Lee and D. G. Messerchmitt, "Synchronous data flow," IEEE Proceedings 37(11), November 1987.

[18] E. A. Lee, "Static scheduling of data flow programs for DSP," Chapter 19 in *Data Flow Computation*, Eds. L. Bic and J-L. Gaudiot, Prentice Hall, Engelwood Cliffs, NJ, 1990.

[19] E. A. Lee, "Consistency in Data Flow Graphs," IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 2, April 1991.

[20] C. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Engelwood Cliffs, NJ, 1985.

[21] C. Hoare, "Communicating Sequential Processes," ACM, Vol. 21, pp. 666-677, Aug. 1978.

[22] J-L. Gaudiot, "Data-driven multicomputers in digital signal processing," IEEE Proceedings, 75(9), pp. 1220-1234, Sept. 1987.

[23] S. Y. Kung, *VLSI Array Processors*, Prentice-Hall, Engelwood Cliffs, NJ, 1988.

[24] T. E. Williams and M. A. Horowitz, "A Zero-Overhead Self-Timed 160ns 54b CMOS Divider," IEEE International Solid-State Circuits Conference, Feb. 1991.

[25] A. P. W. Böhm, *Dataflow Computation*, CWI Tract 6, 1985.

[26] J. C. Ebergen, *Translating Programs into delay-insensitive circuits*, CWI Tract 56, 1989.

[27] E. Brunvand and R. R. Sproull, "Translating Concurrent Programs into Delay-Insensitive Circuits," Proceedings of the International Conference on Computer-Aided Design, April 1989.

[28] A. J. Martin, "Compiling communicating process into delay-insensitive vlsi circuits," Distributed Computing, Vol. 1, No.4, pp. 226-234, 1986.

[29] T-A. Chu, "On the Models for Designing VLSI Asynchronous Digital Systems," Integration, the VLSI Journal, Vol. 4, pp. 99-113, 1986.

[30] T-A. Chu, *Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, MIT, June 1987.

[31] T-A. Chu, et. al., " A Design Methodology for Concurrent VLSI Systems," IEEE International Conference on Computer Design, VLSI in Computers, pp. 407-410, Oct. 1985.

[32] T. Meng, *Asynchronous Design of Digital Signal Processing Architectures*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, U. C. Berkeley, Nov. 1988.

[33] C. J. Tan, *Synthesis of Asynchronous Sequential Switching Circuits*, Ph. D. Thesis, Columbia University, 1969.

[34] D. L. Dill, *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits*, Ph. D. Thesis, Department of Electrical Engineering and Computer Science, MIT, June 1989. An ACM Distinguished Dissertation.

[35] S. M. Novick and D. L. Dill, "Automatic Verification," Chapter 7 in Meng's book *Synchronization Design for Digital Systems*, Kluwer Academic Publishers, 1991.

[36] D. R. Coelho, *The VHDL Handbook*, Kluwer Academic Publishers, 1989.

[37] D. R. Coelho, "Follow Simple Rules to Create VHDL Models," Electronic Design, June 1990.

[38] Lisa Gunn, "VHDL: An EDA Standard Slowly EMERGEs," Electronic Design, June 1980.

[39] R. Lipsett, et. al., *VHDL: Hardware Description and Design*, Kluwer Academic Publishers, Boston, 1989.

[40] L. Augustin, et. al., *Hardware Design and Simulation in VAL/VHDL*, Kluwer Academic Publishers, Boston, 1989.

[41] P. J. Ashenden, *The VHDL Cookbook*, First Edition, Class Notes, Department of Computer Science, University of Adelaide, 1990.

[42] View*logic* ® VHDL Reference Manual, View*logic* System Inc., 1991.

[43] J. F. Wakerly, *Digital Design Principles and Practice*, Prentice-Hall, 1990.

[44] E. J. McCluskey, *Logic Design Principles*, Prentice-Hall, 1986.

[45] F. P. Prosser and D. E. Winkel, *The Art of Digital Design*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[46] W. I. Fletcher, *An Engineering Approach to Digital Design*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

[47] R. E. Miller, *Switching Theory*, John Wiley and Sons, Inc., New York, 1965.

[48] D. B. Armstrong, et. al., "Design of Asynchronous Circuit assuming Unbounded Gate Delays," IEEE Transactions on Computers, Vol. C-18, (12), pp. 1110-1120, Dec. 1969.

[49] C. H. Lau, et. al., "Data Flow Approach to Self-timed Logic in VLSI," ISCAS, 88 pp. 479-482.

[50] L. G. Heller and W. R. Griffin, "Cascode Voltage Switch Logic: A Differential CMOS Logic Family," 1984 IEEE ISSCC Digest of Technical Papers, Feb. 1984.

[51] S-L. Lu, "Implementation of Iterative Networks with CMOS Differential Logic," IEEE Journal of Solid-State Circuits, Vol. 23, No. 4, pp. 1013-1017, August 1988.

[52] S-L. Lu and Milos Ercegovac, "A Novel CMOS Implementation of Double-Edge Triggered Flip-Flops," IEEE Journal of Solid-State Circuits, Vol. 25, No. 4, pp. 1008-1010, August 1990.

[53] S-L. Lu and Milos Ercegovac, "Evaluation of Two Summand Adders in CMOS Differential Logic," IEEE Journal of Solid-State Circuits, Vol. 26, No. 8, pp. 1152-1160, August 1991.

[54] S-L.Lu and L. Merani, "Micro Data Flow", The Fifth Annual IEEE ASIC Conference and Exhibit, ASIC '92, Rochester, New York, 1992.

[55] L. Merani and S-L. Lu, "A Self-timed Approach to VLSI Digital Filter Design," IEEE Pacific Rim Conference on Communications , Computers and Signal Processing, Victoria, B. C., Canada, May 1993.

[56] L. Merani and S-L. Lu, "A Self-timed Approach to VLSI Digital System Design," Graduate Congress '93, Corvallis, OR, April 1993.

[57] S. Chu and C. S. Burrus, "Multirate Filter Designs Using Comb Filters," IEEE Transactions on Circuits and Systems, vol, CAS-31, pp. 913-924, Nov. 1984.

[58] R. S. Schreier, "A Simple Introduction to Decimators," Class Notes, Oregon State University, Corvallis, OR, January 1993.

[59] D. Pountain, "Computing without clocks," BYTE, January 1993.

[60] C-M. Chang and S-L. Lu, "Micro Data Flow Processors," IEEE Pacific Rim Conference on Communications , Computers and Signal Processing, Victoria, B. C., Canada, May 1993.