HIGH-LEVEL MODELLING OF MICROPIPELINES

A thesis submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Science

October 1992

By

Sun-Yen Tan

Department of Computer Science

Contents

List of Figures

List of Tables

Abstract

Asynchronous circuits have the potential to overcome the problems which are encountered in synchronous designs, such as clock distribution and skew. The design of asynchronous circuits has evolved using a modular approach, where a system is designed as an interconnection of modules. "Micropipelines", as expounded by Sutherland, are composed from a particular set of event-driven asynchronous self-timed modules. Their operation is based on a two phase bundled data convention. Such micropipeline design methodologies can simplify the design and reduce the design time and cost.

A micropipeline simulator is required to develop and evaluate micropipeline designs. The implementation of such a micropipeline simulator is presented in this thesis. This implementation involves the construction of Petri net models of the simulated networks, the design of C++ classes for representing circuit models and Petri net models, the definition of notations for entering the simulated micropipelines and describing the simulation, and the design of the simulation procedure and evaluation rules.

Several micropipeline examples are tested to demonstrate that the simulator works correctly. Comparisons of the simulator with a standard hardware simulator, Silos II, are also presented along with an analysis of the performance of the micropipeline simulator and a discussion of the problems encountered during the implementation.

DECLARATION

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Acknowledgements

I wish to thank my supervisor, Professor S. B. Furber, for his continuing guidance and encouragement during this work. This help and advice offered by other members of the AMULET group is also greatly appreciated.

Preface

The author graduated from the National Taipei Institute of Technology of Taiwan in 1982. Between 1984 and 1991 he worked for the same institute as a teaching assistant and was in charge of the senior class's topics of Interface and Peripheral Practice and research projects. In October 1991, the author commenced full-time research work for the degree of Master of Science. The research done during this period is presented in this thesis.

Chapter 1

Introduction

Today semiconductor and VLSI techniques are advancing rapidly. However, chip designs with high performance using synchronous design techniques are constrained by their global clocks because of problems such as clock skew, worst-case delays, and the cost of broadcasting clocks. Asynchronous circuits have the potential to overcome these problems. They also offer several benefits, such as ease of design, low power consumption, and high performance. It is believed that a computer processor can achieve high speed and low power dissipation if implemented with asynchronous circuit design techniques. Therefore, there has recently been renewed interest from research groups in this area of logic design. The first asynchronous microprocessor has been designed by Martin [Martin 89]. Based on their advantages, asynchronous circuits will be in widespread use in computer system design in the near future.

The design of asynchronous circuits has evolved using the modular approach, where a system is designed as an interconnection of modules. In the 1988 Turing Award Lecture, Sutherland expounded a modular approach to building hardware systems based on data-driven asynchronous self-timed logic elements called micropipelines [Sutherland 89]. This micropipeline design methodology has attracted the attention of researchers, particularly in the department of Computer Science at Manchester University. The modular approach reduces the design time

and cost. It is being used to develop high performance and low power microprocessing systems by the AMULET group at Manchester University.

In this research work, this modular approach is considered. The motivation, objectives, and designs of the experiment of this research work will be described in the next section. An overview of this thesis will be given in the last section of this chapter.

1.1 Modelling of micropipelines

1.1.1 The motivation for this research work

When using event-driven logic modules to design micropipelines, an environment for simulating micropipeline designs is needed in order to study the interactions between the event-driven logic modules and to check the correctness of the micropipelines. If the micropipelines are simulated using other simulators, consideration must be given to how each event-driven logic module is designed at gate or transistor level and how to enter the test patterns. Such issues will increase the simulation time and difficulty. Designing micropipelines using event-driven logic modules is like devising a flow chart of a program, which is quite easy. However, designing event-driven logic modules is not easy. Hazard problems and race conditions must be prevented. To simplify the simulation work and to reduce the simulation time, a micropipeline simulator is required.

Modelling techniques are necessary for studying micropipelined asynchronous circuits in order to evaluate high-level specifications, to synthesize low level circuits, and to simulate the circuits after synthesis. The study of this modelling technique is very useful to understand the behaviours of micropipelines and each of the event-driven logic modules. The results of this research work are useful to develop modelling techniques and study mathematical relationships which

can support the micropipeline design methodology to build correct and optimum asynchronous circuits.

1.1.2 The objective of this research work

The goal of this project is to model event-driven logic modules using a standard high-level language and to implement a micropipeline simulator. Micropipeline designs will easily be simulated by such a simulator and the flow of events through the simulated micropipeline circuits can be observed. The test patterns can conveniently be entered to the simulated network and the simulation results can be read and saved into an output file for further analysis.

1.1.3 The experiment design

The experiment design of this project is described as follows:

1. Petri nets are used to model the behaviour of each event-driven logic module. They are also used to control the simulation execution.

2. C++ classes are designed to represent the simulated circuits and models of Petri net.

3. The definitions for entering micropipeline circuits and describing the simulation are made.

4. The steps of constructing the corresponding Petri net model of the simulated network are defined.

5. The simulation procedure is designed.

6. Some micropipeline circuit examples are used to test the designed micropipeline simulator.

7. Some micropipeline circuit examples are simulated using the Silos II standard hardware simulator.

8. Both simulation results are compared and discussed.

9. Finally, the implementation and the modelling technique are summarized in the design report.

1.2 An overview of this thesis

The motivation, objective and the experiment design of this research work and an overview of this thesis are given in this chapter. Chapter 2 will describe some basics of asynchronous systems, classification, advantages and disadvantages of asynchronous circuits, and characteristics of micropipelines. The two-phase bundled data convention and event-driven logic modules will also be discussed. A brief introduction to Petri nets, and some modifications to the standard firing rules of Petri nets, will be described in Chapter 3. Next, the properties and power of Petri nets will be examined. Then the Petri net models of event-driven logic modules will be described. Chapter 4 will define a temporary notation for describing micropipeline circuits and the simulation work for this research stage and present the C++ classes which will be used to represent the circuit model and the Petri net model. Chapter 5 will present the design and operation of the micropipeline simulator. The method of constructing the corresponding Petri net model of the simulated network will be explained. The simulation procedure, which will be used to control the simulation work concurrently, will also be presented in Chapter 5. Then several micropipeline examples will be tested to demonstrate that the simulator can work correctly. In Chapter 6, the analysis of the performance of the micropipeline simulator, the problems which were encountered during the implementation, how common errors of micropipelines are

discovered when the simulator executes, and a comparison of the simulator with a standard hardware simulator, Silos II, will be presented. Then the advantages and disadvantages of the micropipeline simulator will be summarized. Finally, Chapter 7 will give a short conclusion and proposals for further research.

Chapter 2

Asynchronous design

Traditional sequential circuits have a global clock to control all the functional units operating in lock-step. Recently asynchronous design has been the subject of increasing interest because the system clock causes serious difficulties in the design of high-performance chips. One limiting factor on the maximum clock rate is clock skew. A fully asynchronous design does not need a global clock and there is no problem of clock skew. Although it is more difficult to design asynchronous circuits than synchronous circuits, some researchers have presented modelling techniques [Peterson 81], new design methodologies [Sutherland 89], and synthesis techniques [Meng 89] for analysing asynchronous behaviour, which have encouraged other researchers to develop asynchronous circuits. The first design of an asynchronous processor was published by Martin [Martin 89].

Asynchronous logic circuits have several important advantages over their counterparts in clocked logic. An asynchronous logic function is potentially faster because it works at the average-case delay rather than the worst-case delay. There is no global clock on asynchronous circuits so that they will not unnecessarily dissipate power when there is no useful work to do. Therefore, asynchronous logic can be used to implement systems with lower power dissipation.

In this chapter, the basics of asynchronous systems will first be introduced briefly. An overview of the current state of asynchronous design methodologies

will then be given, paying particularly attention to the approach expounded by Sutherland in his 1988 Turing Award lecture.

2.1 Some basics of asynchronous systems

Digital circuits may be split into one of two classifications: synchronous circuits and asynchronous circuits. Circuits are called asynchronous if they do not contain a global clock. The operation of a synchronous system is based on a global clock. Each storage element and subcircuit within synchronous circuits are triggered by this global clock, i.e., the timing and sequence of synchronous circuits is controlled by this global clock. The clock period must be greater than the delay of the slowest combinational path in an entire system when the computation progresses. Some problems are encountered when using synchronous circuit design with a global clock:

1. A synchronous circuit is divided into several subcircuits. Each subcircuit is triggered by the global clock to perform a subcomputation. The clock period must be larger than the worst-case delay of any subcomputation. Therefore, synchronous circuits always operate at the worst-case conditions.

2. A clock pulse usually triggers a lot of subcircuits. It is necessary to consider the driver current and load on the clock driver circuitry. For example, Greenstreet has pointed out that the clock pulse may completely disappear within a very long pipeline [Greenstreet 88].

3. It may be necessary to re-analyse the timing and delays of the entire system, or to re-design completely the system if a functionally equivalent but better subcircuit is replaced in a system.

Asynchronous circuits are not clock driven but event driven, where an event is the completion of subcomputations or operations. Subcomputations in asynchronous circuits are triggered by completion signals of other subcomputations rather than by a central clock. Therefore, asynchronous circuits have the potential for overcoming many of the problems with synchronous circuits.

Asynchronous circuits have been studied for almost as long as digital computers exist. Although asynchronous circuits can overcome those problems with synchronous circuits, most computer systems are still synchronous. Recently some functional building blocks for events have been published [Sutherland 89]. However, it is more difficult to design asynchronous circuits which are hazard-free and have no race conditions. Asynchronous systems can consist of such building blocks. The design of asynchronous systems becomes easier than before.

Each subcircuit of an asynchronous circuit can operate at its own rate. The limiting factors of their operation are only the abilities of their neighboring subcircuits which produce their input data and consume their output data. Therefore, a local communications protocol is required when a subcircuit transmits its information to such neighbouring subcircuits or receives some information from them. With such a local communication protocol, a transmitter subcircuit must indicate that the data is available (i.e. a request signal) and a receiver subcircuit must reply that the data has been accepted (i.e. an acknowledge signal). Asynchronous computation starts the next task cycle once the current task is completed. Since the completion time of each asynchronous operation is task-dependent, an "average" processing speed can be achieved as opposed to the worst-case speed in synchronous designs [Meng 89]. A modular design approach, average-case delays and no global clocks can result in an asynchronous system design with low cost, faster speed, and low power consumption. A more detailed definition of an asynchronous system is given in [Gopalakrishnan 90].

2.2 Classification of asynchronous circuits

Asynchronous circuits can be categorised into several classes by their features and architectures. A brief description of such a classification scheme is given in the following subsections.

2.2.1 Delay-insensitive circuits

Delay-insensitive circuits are a special type of asynchronous circuits composed of protocol-based modules, i.e., a circuit is delay-insensitive if the correct operation of the circuit does not depend on any assumptions about the delay in wires or the operators of the circuit. Such circuits do not use a clock signal or knowledge about delays [Martin 89], which means that there is no assumptions about delays for delay-insensitive circuits within the system. Any gate or interconnection may take an arbitrary time to propagate a signal. This property guarantees that any correctly functioning subcircuits may be composed together and continue to operate correctly [Brunvand 89]. Therefore, delays-insensitive techniques are particularly attractive for VLSI synthesis [Martin 91].

The use of delay-insensitive circuits offers several advantages over that of synchronous circuits. Delay-insensitive circuits are more robust and potentially faster than their clocked counterparts, since their correct operation does not depend on the worst-case delay assumptions. A delay-insensitive circuit can be formally derived by program-transformation from a high-level program description [Martin 92].

2.2.2 Self-timed systems

A self-timed system [Seitz 80] is either a self-timed element or a 'legal' interconnection of self-timed systems. Self-timed elements must be contained within an

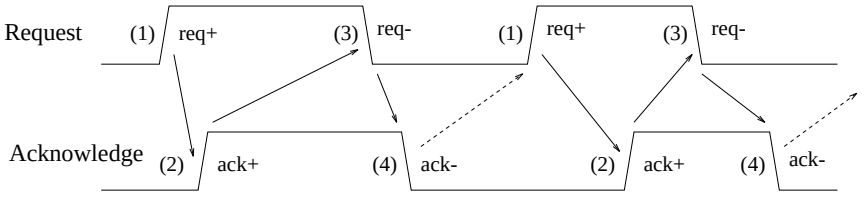
equipotential region, i.e., a region in which wire delays are negligible. Within such regions, self-timed data communication is implemented by means of a data-valid wire. For self-timed communication between equipotential regions, where wires may introduce arbitrary delays, two or four-phase handshaking protocols, which will be discussed in subsequent subsection, are used. Therefore, it is assumed that a circuit can be decomposed into equipotential regions inside which delays in wires are negligible if self-timed techniques are considered [Seitz 80]. Self-timed circuits are delay insensitive, i.e., their behaviour does not depend on any relative delays among physical elements [Unger 69] [Seitz 80] [Meng 89].

2.2.3 Speed-independent circuits

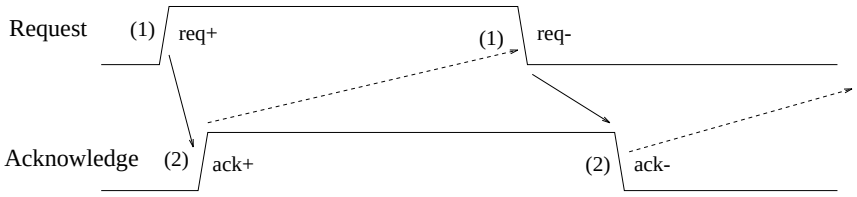
Speed-independent techniques assume that delays in gates are arbitrary, but there are no delays in wires [Martin 89]. Therefore, speed-independent circuits adhere to a less strict model, i.e., gates may have an arbitrary propagation delay but transmission along wires is instantaneous. Some issues of the modelling, specification, and verification of speed-independent circuits are illustrated through the consideration of self-timed queues in [Dill 92]. There are no race conditions and hazards within speed-independent circuits and such circuits operate at maximum speed (i.e. they allow two or more signals to be excited simultaneously.). More detail about the advantages and disadvantages of speed independent circuits is given in [Miller 65].

2.2.4 Four-phase handshaking protocol

Two systems may communicate using a four-phase handshaking protocol [Seitz 80] as shown in Figure 2.1(a). The two control wires which are used to indicate that the data has been available (i.e. request signal or req) and the data has been accepted (i.e. acknowledge signal or ack) will return to their initial state after



(a): Four-phase handshaking protocol



(b): Two-phase handshaking protocol

Figure 2.1: Four-phase and two-phase handshaking protocols

they have activated. This handshaking protocol always uses the rising transitions to initiate operation and the falling transitions to reset signals. The operating cycle is (1) request active, (2) acknowledge active, (3) request inactive and (4) acknowledge inactive, i.e., the cycle is $req+$; $ack+$; $req-$; $ack-$. The request signal adopting its active level indicates that data is ready for transfer. The acknowledge signal adopting its active level indicates that the transfer is complete. When the sender detects that the transmission has finished, it will inactivate its request signal. Then the receiver will inactivate its acknowledge signal. Therefore, the four-phase handshaking protocol is also called return-to-zero signalling. Most asynchronous systems use this protocol to synchronize operations between neighbouring subsystems.

2.2.5 Two-phase handshaking protocol

In a two-phase handshaking protocol [Seitz 80] which is shown in Figure 2.1(b), there are also two control wires. One is used to indicate the data is available (i.e. request signal or req). The other is used to indicate the data has been accepted (i.e. acknowledge signal or ack). In this signalling scheme, rising transitions and falling transitions of the control wires have the same meaning. They are called the request event or the acknowledge event. The phases of the two-phase handshaking protocol are: (1) request event and (2) acknowledge event. For example, $req+$; $ack+$ is an active two-phase cycle. The next cycle for this interface would be $req-$; $ack-$. The signals do not need to return to their initial state after they have activated. The opposite transition has the same meaning and may be used to represent the subsequent event. If the request wire goes to a 'logic high' level for representing a current request event, the next request event will be represented by the wire going to a 'logic low'. In this mechanism, it is the occurrence of a transition rather than a level which carries the information. Therefore, two-phase signalling is also called non-return-to-zero and transition signalling or event signalling. Transition signalling is potentially useful as it allows the transmission of information without requiring that signals return to their initial state after each event. This removal of redundant signal transitions improves the efficiency of the communication system with consequential higher performance. This is also pointed out in [Williams 90]. The circuits working on this protocol have low power dissipation. Some event-driven logic modules are presented in [Sutherland 89] which can be used to compose control circuits easily. Also there is a four-phase to two-phase converter with "quick return" which has been published in [Gopalakrishnan 90].

2.2.6 Double-rail encoding

In this encoding system, an *n-bit* data value is denoted by 2^n+1 code combinations on $2n$ wires. It is used to indicate that every bit of a data word has arrived at a defined logic stage before the evaluation can commence within a truly delay-insensitive circuit. This means that the timing information must be encoded with the data. The usual way of achieving this is to use two separate wires to represent each bit of data, one wire for transmitting the logic "0", the other for transmitting the logic "1". The arrival of a bit is then denoted by a transition on the relevant wire. The arrival of each bit can be detected using an XOR gate and the arrival of the entire word can be detected by the arrival of all the data bits in the word. This detection can be implemented by a Muller C-element. Because this encoding technique requires two wires for each data bit, designs using the technique require about double the chip area needed by an equivalent synchronous circuit.

2.3 Advantages and disadvantages

Asynchronous circuits have the potential to overcome some fundamental problems encountered in synchronous circuit design. The possible advantages are as follows:

1. New synthesis techniques and design methodologies of asynchronous circuits have been studied and published, and in some cases the design cost and time are explicitly reduced.

2. There is no global clock for synchronizing the operation of each subcircuit. Consequently, asynchronous circuits have better composability. They more readily allow concurrency due to the absence of global clocks.

3. Asynchronous circuits have low power dissipation. Clock drivers may dissipate most of the power of a synchronous circuit. Power consumption may

be a significant cost factor due to the costs of packaging for high power dissipation.

4. Some basic modules which can easily be used to compose asynchronous circuits have been published [Sutherland 89]. The high-level design of asynchronous systems is then like constructing flow diagram of a program, which is relatively straightforward.

5. Improving the performance of an asynchronous system can be achieved by replacing a critical subsystem. Re-designing and re-analysing the entire system is not necessary.

6. Asynchronous circuits tend to reflect the average-case delay rather than the worst-case delay. However, the design style may slow down the operating speed of the circuit. For example, the speed of a circuit which uses a four-phase handshaking protocol is usually slower than one using a two-phase handshaking protocol, because the return-to-zero of the four-phase handshaking protocol increases the operating cycle.

7. Error detection is not difficult in asynchronous circuits, since an asynchronous circuit which contains an execution sequence error usually deadlocks.

There are also some disadvantages as follows:

1. Asynchronous circuits need more wiring for handshaking between neighbouring subcircuits. Therefore, asynchronous circuits usually need a greater silicon area.

2. Low-level asynchronous design is harder and more constrained than synchronous design, due to the hazard problem and race conditions.

Sutherland in his 1988 Turing Award lecture [Sutherland 89] discusses a class of asynchronous circuits called "micropipelines". Here, circuits are designed using a set of functional modules. This greatly simplifies the design process. Micropipelines will be described in more detail in the next section.

2.4 Micropipelines

Pipelining [Hennessy 90] is an implementation technique whereby multiple instructions or data are overlapped in execution. A pipeline is like a production line, where each step in the pipeline completes a part of the instruction. Each such step is called a pipe stage or a pipe segment. The throughput of the pipeline is a measure of how many instructions or data can pass through it per unit time. The latency of a pipeline is a measure of how long it takes a single instruction or data to pass through it. When designing pipelines, both the latency and throughput are important. The time taken for an individual instruction or data item to traverse the pipeline, i.e. the latency, must be minimised. The number of instructions or data items processed in a given time i.e., the throughput, must be maximised. A synchronous pipeline consists of several stages which operate in lockstep. There is a global clock which is used to define when all stages must have completed their operations and data is ready to be transferred to the subsequent stages. The minimum clock period of the pipeline is determined by the time taken for the slowest pipeline stage to complete its operation. The latency of the pipeline is the clock period multiplied by the number of pipeline stages. Therefore, the performance is limited by the slowest stage.

Some research into asynchronous pipelines has shown that there is the potential for high performance and concurrency in asynchronous pipelines [Komori 88] [Ginosar 90] [Williams 90]. There is no central clock in an asynchronous pipeline.

Each stage operates at its own rate. Each stage only needs to wait for the operation of the neighboring stages to be complete. The operation of most asynchronous pipelines is based on a four-phase handshaking protocol. Micropipelines are members of a class of asynchronous pipelines whose operation is based on a two-phase bundled data convention and are self-timed, event-driven systems [Sutherland 89]. Their operation is faster than a four-phase protocol would normally allow. Before introducing micropipelines, a brief description of the two-phase bundled data convention will be given.

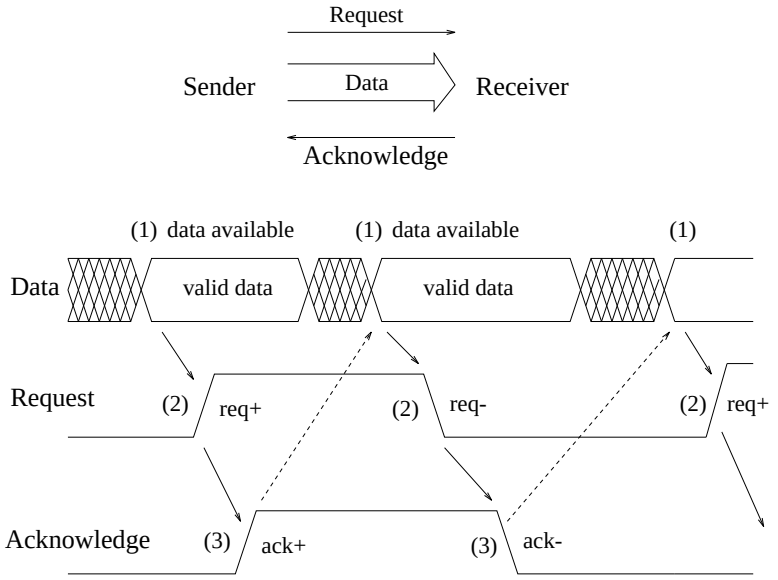


Figure 2.2: Two-phase bundled data convention

2.4.1 The two-phase bundled data convention

A "two-phase bundled data convention" is a communications system where a two-phase handshaking protocol is used and an arbitrary number of data wires must be treated as a bundle together with the request signal wire [Sutherland 89]. In the two-phase handshaking protocol, rising transitions and falling transitions

of either control wire have the same meaning, i.e., they represent request events or acknowledge events. In this signalling scheme, the operating cycle is (1) data available (2) request event, and (3) acknowledge event. The data signals can use a double-rail encoding or a traditional data representation which is similar to that used in synchronous circuits, such as the *8-4-2-1 code* ... etc. Figure 2.2 illustrates this signalling. When using a two-phase bundled data convention for transmitting data between two systems, it must be noted that the data signal delays must be no longer than the request signal delays to the point where the data signals and the request signal arrive at the receiver.

2.4.2 Event-driven logic modules

Various circuits have been devised for controlling transition signals. These are called event-driven logic modules in [Sutherland 89]. They are used in next section to construct the control circuits of micropipelines. Figure 2.3 shows the block diagrams of such event modules.

Muller C-element: A C-element [Miller 65] performs the rendezvous function. A Muller C-element is shown in Figure 2.3(a). It is a very important element in asynchronous circuit designs. When both of its inputs are in the same logical state, i.e. it has received an event on both of its inputs, its output will have the the same state as the inputs, i.e. there is an event on its output. When the two inputs are different, i.e. only one event has arrived on either of its two inputs, it retains its previous state and does not change its output, i.e. there is no event on its output. Thus only when both of its inputs have received an event, will an event appear at its output. For three or more inputs, it is required that all of them receive an event before producing an event at its output. Therefore, such elements are AND elements for events. Such elements can be used to treat the forking and joining connection problem of circuits.

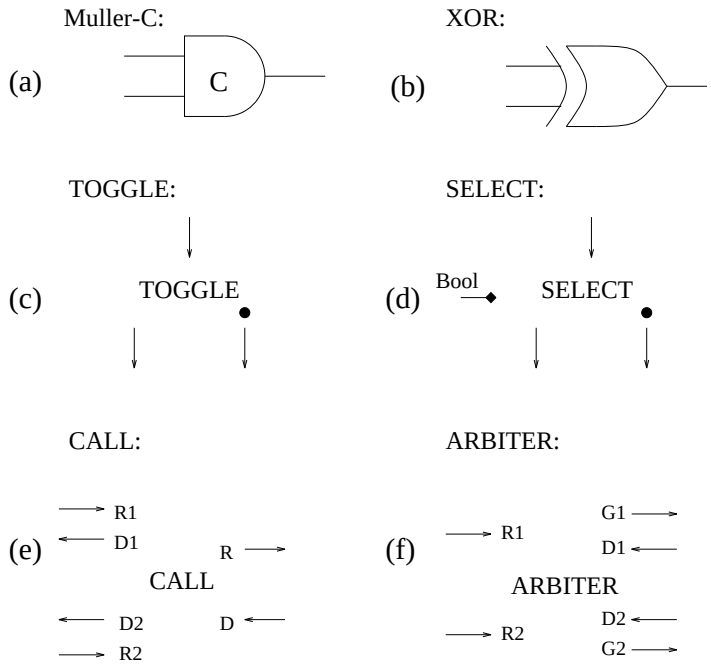


Figure 2.3: Block diagrams of event-driven logic modules

XOR circuit: An XOR circuit performs the merge function for events. It is shown in Figure 2.3(b). Such circuits are also called the OR element for events. When one of its input changes state, its output also changes state. Therefore, when it receives an event on either of its two inputs, it will produce an event on its output. For more than two inputs, XOR performs the multiple-input OR function for events. Such elements can be used to treat the uncertain source connection problem of circuits.

TOGGLE circuit: A TOGGLE circuit is shown in Figure 2.3(c). After the master clear signal activates, the odd events which arrive on its input will be sent to its dotted output and the even events which arrive on its input will be sent to its non-dotted output . Such elements can be used to treat the regularly alternate connection problem of circuits.

SELECT module: A SELECT module is shown in Figure 2.3(d). An incoming event which arrives on its input will be sent to the output labelled "true" or the output labelled "false", depending on the value of a data input. The data input is a Boolean value. If the data is true, the incoming event is sent to the the output labelled "true". If the data is false, the incoming event is sent to the the output labelled "false". The Boolean value must arrive before the incoming event. Therefore the Boolean data and the incoming event must be treated as a bundle. Such elements can be used to treat the selective connection problem of circuits.

CALL module: A CALL module is shown in Figure 2.3(e). It remembers the event which is most recently received on its input and correctly returns an event on the output corresponding to the incoming event when the called procedure has finished. We need to note that the CALL module will operate properly only if each call completes before a subsequent call occurs. For this reason, we need an element to treat the concurrency problem and to ensure that the subsequent event does not happen before the previous call has finished. In general, such elements can be used to treat the procedure connection problem of circuits.

ARBITER module: In asynchronous pipelines, if two events from two source pipes need to join into a single pipe without strict ordering, i.e. such two events occur simultaneously or nearly simultaneously, an arbitration element must be used to decide the ordering of the events. It is such an important element in asynchronous pipeline circuit designs that there has been much research into the design of such modules [Plummer 72] [Pearce 75] [Calvo 86]. An ARBITER module is shown in Figure 2.3(f). The sequence of two events without strict ordering can be clearly decided, and then a grant event for only one of them will be produced on the corresponding output.

It delays subsequent grants until that the earlier grant has been finished. Therefore, only one grant at a time is sent. An ARBITER element can be connected directly to the CALL element to treat the arbitration problem. Arbiter trees can be used to implement arbiters which have more than two inputs [Plummer 72].

2.4.3 A micropipeline stage

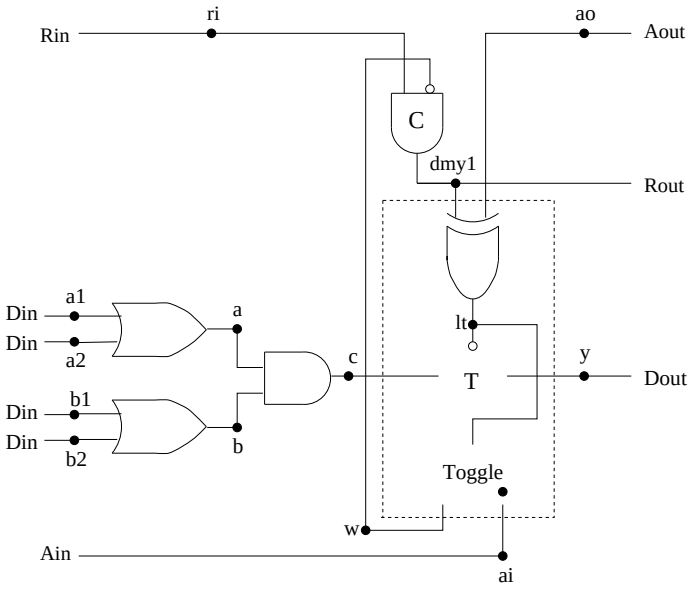


Figure 2.4: A micropipeline stage

Figure 2.4 is an example of a micropipeline stage. A micropipeline stage consists of control circuits, combinational logic circuits and storage elements. Its operation is based on the two-phase bundled data convention with rising transitions and falling transitions of control wires having the same meaning. Both transitions are called events and could represent request signals or acknowledge signals. When the data is available on the inputs, there is an event called "request" on the "Rin" to enable the storage elements to catch this valid data. After the storage elements hold the predecessor's data, there is an event called

"acknowledge" on the "Ain" to inform the predecessor stage that the data has been accepted and an event called "request" on the "Rout" to enable the successor stage to catch the data. When the successor stage holds data, it will place an event on the "Aout" to clear the storage elements of the current stage and enable the current stage to catch the subsequent data.

The operation cycle of micropipelines is (1) data available, (2) request event, and (3) acknowledge event. The return inactive state action is not necessary. Therefore, the operating speed of micropipelines is faster than that of four-phase handshaking protocol pipelines. The control circuits are composed from event-driven logic modules, such as muller-C, toggle, select, call ... etc. In this example, the combinational logic circuit consists of two OR gates and one AND gate. The storage element consists of an XOR, a low-activated transparent latch and a TOGGLE. The control circuit here is only a dmuller-C. It is used to implement the rule "if predecessor and successor differ in state then copy predecessor's state else hold present state" [Sutherland 89]. This rule allows the data to flow through the micropipeline stages. Such event-driven logic modules can also be used to connect micropipeline stages in different configurations. For example, Figure 2.5 shows how five micropipeline stages are connected in series. Figure 2.6 shows a micropipeline stage that forks into two micropipeline stages. Figure 2.7 shows a micropipeline stage that forks into two micropipeline stages which subsequently join into one micropipeline stage. Figure 2.8 shows a micropipeline stage which selectively connects to one of the two micropipeline stages which call the same micropipeline stage.

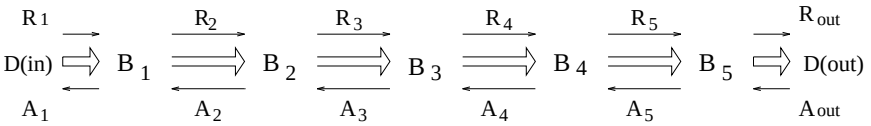


Figure 2.5: Five serially connected micropipeline stages

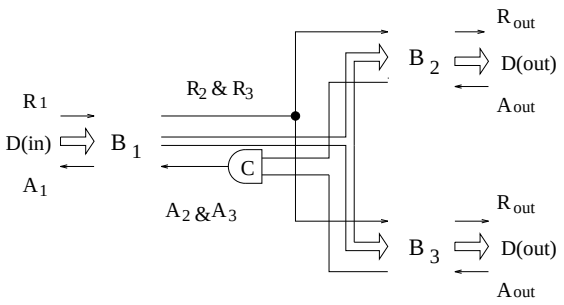


Figure 2.6: A micropipeline stage forks into two micropipeline stages

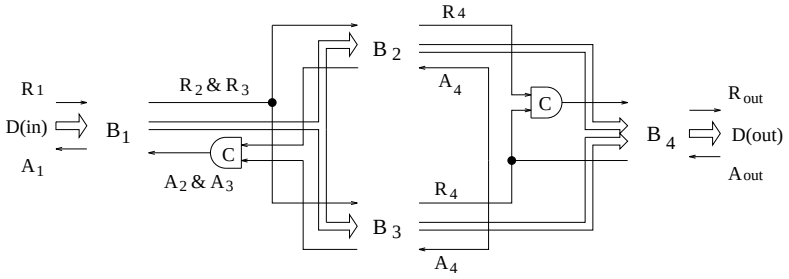


Figure 2.7: A micropipeline stage forks into two micropipeline stages which join into one micropipeline stage

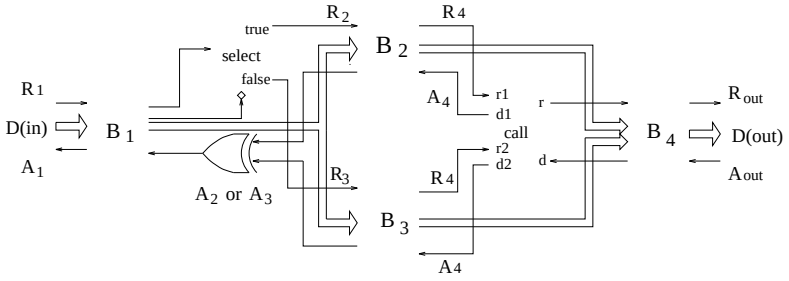


Figure 2.8: One micropipeline stage selectively connects to one of two micropipeline stages which call the same micropipeline stage

2.4.4 Properties of micropipelines

Micropipelines are one kind of particularly simple and elastic pipeline in which computations can be implemented by combinational logic functions. If a micropipeline has no computation in it, it is a FIFO (Figure 2.9). Inserting some FIFOs in between the stages to accommodate waiting instructions can improve the performance of pipelines.

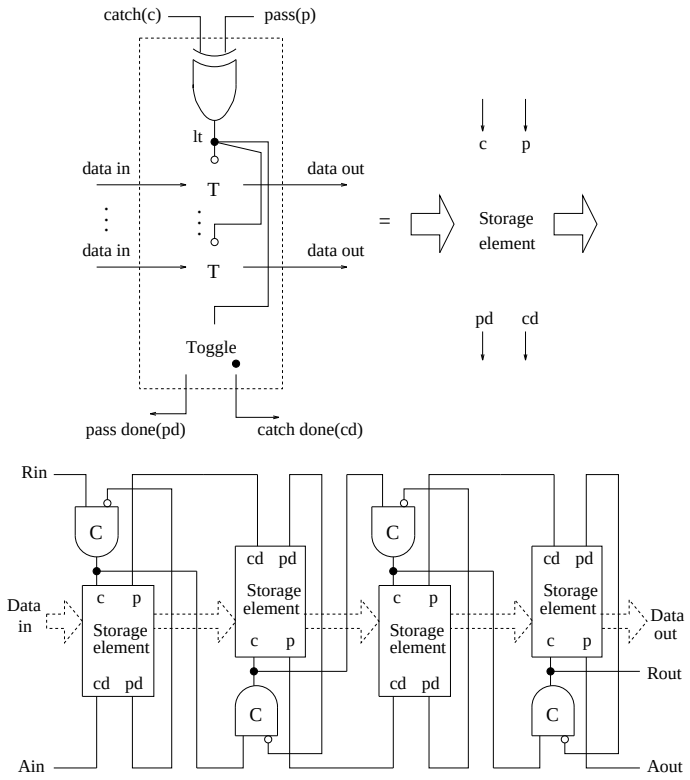


Figure 2.9: A micropipeline FIFO

Within such a FIFO, if the states of all of the Muller C-element which are used to control the storage elements have both their inputs in different states, this situation indicates that the FIFO is empty. On the other hand, if the states of all such Muller C-element are such that both their inputs have the identical state, this situation indicates that the FIFO is full. To initialize a FIFO or a

micropipeline in the empty state, all inputs of the Muller C-elements within such a FIFO or micropipeline can be set to the same state by a master clear signal.

When a micropipeline with or without computations (FIFO) is full, the occurrence of the acknowledge event must wait until the second stage has consumed the data held in the first stage. After it issues an acknowledge, the subsequent data can effectively enter into it. This means that the input device connected to the micropipeline cannot change its input data until it has received an acknowledge event for the present data. A similar handshake will occur at the output end of the micropipeline. Therefore, correct data with average-case speed can be propagated through the micropipeline.

2.4.5 The advantages of micropipelines

In addition to the advantages of asynchronous circuits, micropipelines have several further advantages as follows:

1. In micropipelines, the use of a two-phase bundled data convention not only simplifies the initial system design but also permits rapid mid-life upgrade of systems as new technology becomes available. Components are easily replaced by new ones. The system can more readily maintain the characteristic of high performance and low cost.

2. Because event-driven logic modules provide conditional operators, procedure calls, and other elements familiar to programmers, this makes such modules easy to compose into loops and other structures similar to those found in programs. The design of control circuits is rather like making block diagrams of programs. Complex functions are easy to compose from simple modules which provide basic functions already familiar in programming. More complex systems can be built by composing a hierarchy of

the basic modules and previously designed compositions. It is very easy to understand and to realize.

3. The micropipeline design methodology reduces the design time and cost of complex asynchronous systems. It provides an exciting alternative to conventional hardware design. The micropipeline design methodology simplifies system design because simple modules and their compositions can be further used to construct large systems.

4. Returning to the inactivate state is not necessary. Such micropipelines save time and energy costs on the return transition. Therefore, when micropipeline techniques and event-driven modules are used to design complex asynchronous systems, the systems have the property of high performance and low power dissipation. These techniques and modules are being used to develop high performance and low power microprocessing systems.

Micropipelines have a few disadvantages as follows:

1. The communication between stages or storage elements in micropipelines is not fully delay-insensitive if control and data signals are both considered. The delays in data transmission must be less than the delays in transmitting the request signal. The combinational logic circuits for computations within the micropipeline must be carefully decided and designed. High speed electronics techniques, such as dynamic CMOS techniques, are applied when designing this part.

2. The data signals must propagate through a micropipeline faster than the control events through its control circuits. Therefore, special delays are sometimes required in the control path when significant processing logic is put between storage elements in the data path.

Chapter 3

Petri net model

The goal of this chapter is to present a Petri net model of each of the event-driven logic modules. A brief introduction to Petri nets, and some modifications to the standard firing rules of Petri nets, will be described. The properties and power of Petri nets will be examined. Finally, Petri net models of the event-driven logic modules will be discussed. Their operation will be explained in detail. In Chapter 5 these Petri net models will be used to control the simulation of the micropipeline models.

3.1 Petri nets

Petri nets [Peterson 77] [Peterson 81] were developed by C. A. Petri in 1962. They are used to describe the flow of discrete events. A Petri net C consists of five parts: a set of places P, a set of transitions T, an input function I, an output function O and a marking function M. It is denoted by $C =< P, T, I, O, M >$ where

$P = \{p_1, p_2, ..., p_n\}$ is a set of places which represent conditions,

$T = \{t_1, t_2, ..., t_m\}$ is a set of transitions which represent events,

$I : T \rightarrow P$ is an input function which defines predecessor places of a transition,

or $P \rightarrow T$ is an input function which defines predecessor transitions of a place,

$O : T \rightarrow P$ is an output function which defines successor places of a transition,

or $P \rightarrow T$ is an output function which defines successor transitions of a place

and

$M : P \rightarrow N$ is a marking function which defines the token number of a place,

$N = \{0, 1, 2, ...\}$.

A Petri net graph G is a bipartite directed multigraph, because a Petri net graph has two types of nodes: places and transitions. A circle and a bar represent a place and a transition respectively. The arcs from places to transitions and from transitions to places represent the input functions and the output functions respectively. The movement of a token inside a place depicts the flow of discrete events.

A Petri net is executed by firing transitions. When a transition is fired, one token will be removed from each of its input places, and each of its output places will be assigned a new token. A transition can be fired if it is enabled. A transition is enabled if there is at least one token inside each of its input places. After firing a transition, the marking of the Petri net will be changed, i.e. firing an enabled transition in a marked Petri net with marking μ will result in the Petri net with a new marking μ'. Transition firings can continue as long as there exists at least one enabled transition. When there are no enabled transitions, the execution halts.

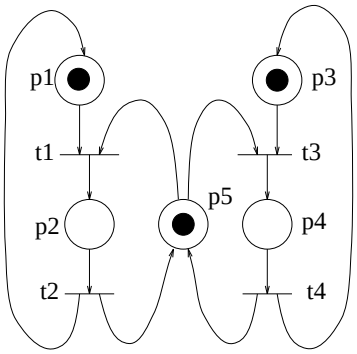


Figure 3.1: A Petri net example

Figure 3.1 shows a Petri net example. In this example, there is a set of places $P = \{p_1, p_2, p_3, p_4, p_5\}$, and a set of transitions $T = \{t_1, t_2, t_3, t_4\}$. The relationship between places and transitions in this Petri net can be denoted by their input and output functions as follows:

$$I(t_1) = \{p_1, p_5\}, \quad I(t_2) = \{p_2\}, \quad I(t_3) = \{p_3, p_5\}, \quad I(t_4) = \{p_4\},$$
$$O(t_1) = \{p_2\}, \quad O(t_2) = \{p_1, p_5\}, \quad O(t_3) = \{p_4\}, \quad O(t_4) = \{p_3, p_5\},$$
$$I(p_1) = \{t_2\}, \quad I(p_2) = \{t_1\}, \quad I(p_3) = \{t_4\}, \quad I(p_4) = \{t_3\},$$
$$I(p_5) = \{t_2, t_4\},$$
$$O(p_1) = \{t_1\}, \quad O(p_2) = \{t_2\}, \quad O(p_3) = \{t_3\}, \quad O(p_4) = \{t_4\},$$
$$O(p_5) = \{t_1, t_3\},$$

The marking of this Petri net can be denoted by the marking function as follows:

$$M(p_1) = 1, \quad M(p_2) = 0, \quad M(p_3) = 1, \quad M(p_4) = 0, \quad M(p_5) = 1.$$

The transitions t_1, t_3 are both enabled. However, either transition t_1 or transition t_3 will be fired. If transition t_1 is fired, transition t_3 will be disabled. Similarly, if transition t_3 is fired transition t_1 will be disabled. This example clearly shows the nondeterministic feature of Petri nets.

Within a Petri net containing a marking μ, a set of transitions will be enabled and may be fired. The result of firing a transition within such Petri net is that the Petri net will contain a new marking μ'. This new marking μ' is called immediately reachable from μ. If μ' is immediately reachable from μ and μ'' is immediately reachable from μ', then μ'' is called reachable from μ.

Petri nets were designed for and are used mainly for modelling. The conditions of a system are modelled by places in a Petri net; the events of a system are modelled by transitions. The holding of a condition is represented by a token in the place corresponding to the condition. When the transition is fired it removes the tokens representing the holding of the precondition and creates new tokens which represent the holdings of the postconditions.

3.2 Additional firing rules

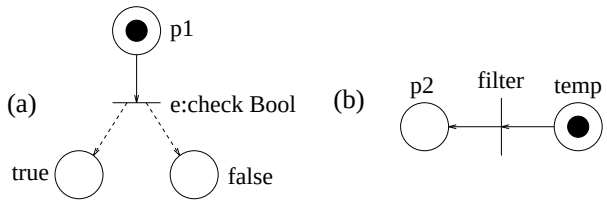


Figure 3.2: The two modified firing rules of Petri nets

The standard firing rule of Petri nets is each output place of the firing transition will be assigned a token if a transition is fired. However, there is an additional firing rule which will be defined here to model the SELECT module (See 2.4.2). The additional rule is that only partial output places will be assigned tokens rather than all output places will be assigned tokens when a transition is fired. Whether the output place is assigned a token or not is decided by the result of the corresponding evaluation of the firing transition. There are two use cases of this additional firing rule. The first use case is shown in Figure 3.2(a). If the transition

labelled "e:check Bool" is fired, the token inside the place p1 will be removed. Either the place labelled "true" or "false" will be assigned one token depending on whether the result of the corresponding evaluation of the transition is true or false. Therefore, the place labelled "true" will be assign a token if the result is true. The place labelled "false" will be assign a token if the result is false. The second use case is shown in Figure 3.2(b). When the transition labelled "filter" is fired, the token inside the place labelled "temp" will be removed. However, the place *p2* will be assigned one token or not depending on the corresponding checking result of the firing transition. The place *p2* will be assign a token if the the corresponding checking result is expected. On the other hand, the place *p2* will not be assign a token if the the corresponding checking result is not expected.

3.3 The power of Petri nets

Particular Petri nets have properties which may help in the analysis of the modelled system. Some important properties are safeness, boundedness, conservation, liveness, reachability and coverability, and equivalence and subset [Peterson 81]. Generating the reachability tree, which represents the reachability set of a Petri net, is the major analysis technique. The above properties may be verified using this analysis technique.

Petri nets can model the inherent parallelism or concurrency of systems. One major feature of Petri nets is their asynchronous nature. Petri nets have been successfully used to model many software and hardware systems. There are examples of the use of Petri nets to model speed independent asynchronous circuits [Misunas 73] [Agerwala 79]. The safeness, liveness, and boundedness properties of Petri nets are used to analysis such circuits. Petri nets have also been used in the performance evaluation of asynchronous concurrent systems [Ramamoorthy 80]. Petri nets can also be used to describe the high-level specification of self-timed

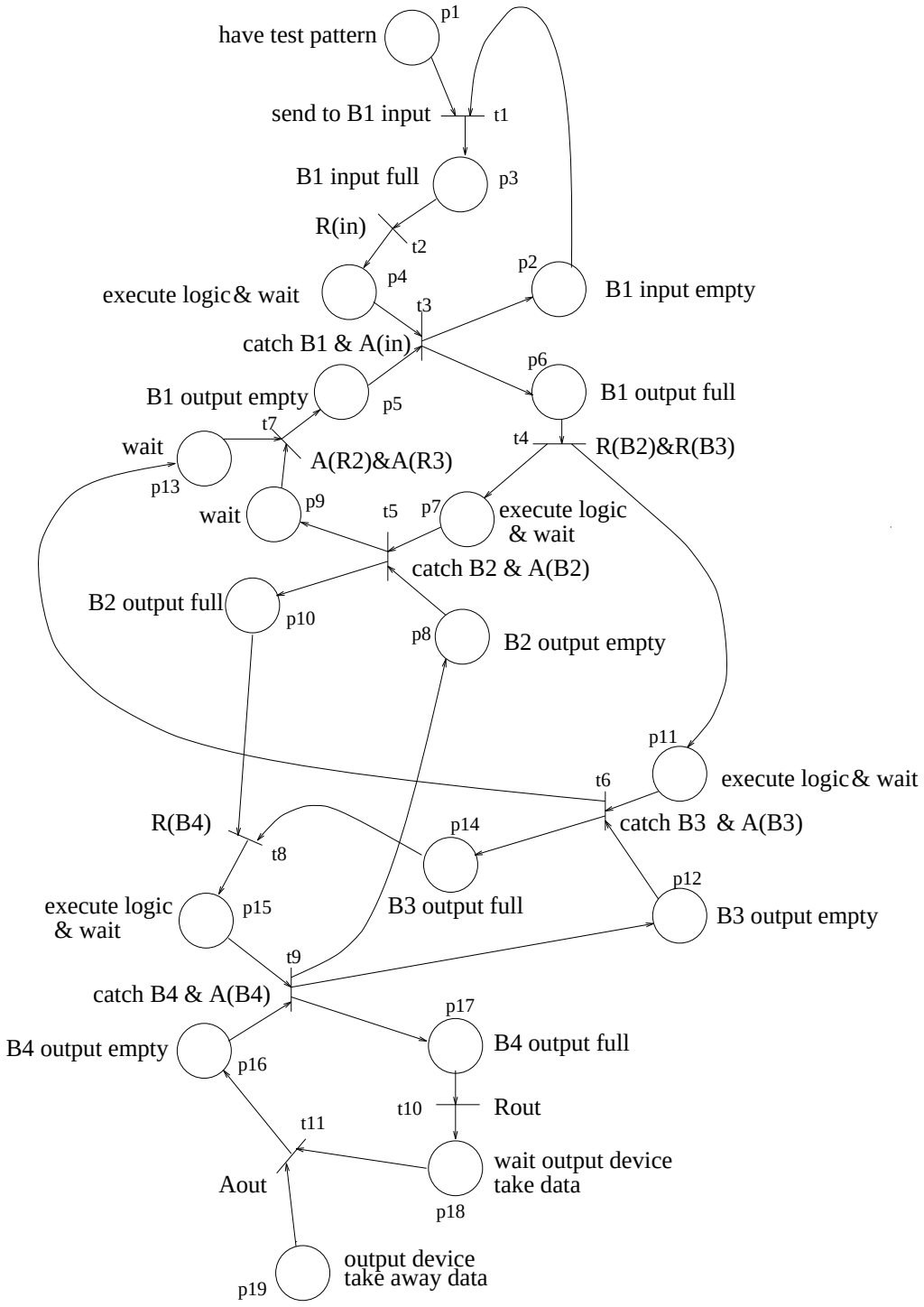


Figure 3.3: An intuitive Petri net model of the circuit of Figure 2.7

queues whose operation is based on a two-phase bundled handshaking protocol [Dill 92]. Modelling a system using Petri nets has three potential advantages: it is easy to understand, to analyse, and to verify. Petri nets are powerful due to their ability to represent concurrency and conflicts.

Some subclasses of Petri nets have also been used in the synthesis of asynchronous networks, such as J-nets presented by Joerg [Joerg 90] and Signal Transition Graphs presented by Chu [Chu 86] [Lavagno 91].

One important reason for choosing Petri nets to model event-driven modules is that many mathematical tools and theories have been developed which can be applied to the asynchronous network [Peterson 81]. Intuitively, the behaviour of the circuit of Figure 2.7 can be describe by the Petri net shown in Figure 3.3. The flow of tokens which represent events (requests or acknowledges) through the network closely models the flow of control signals through the actual circuit.

3.4 Petri net models of logic modules for events

After analysing the behaviour of event-driven logic modules, Petri net models of each event-driven logic module can be constructed.

Muller C-element

The Petri net model of a Muller C-element is shown in Figure 3.4(a). A two-input Muller C-element is an AND and JOIN element for events. When events on both inputs arrive, then there is an event on the output. An event arrival can be denoted by a token inside a place. When both corresponding places of two inputs contain a token, firing the corresponding transition results in the corresponding place of the output containing a token, i.e. an event will appear on the output of the Muller C-element.

XOR

The Petri net model of an XOR is shown in Figure 3.4(b). A two-input XOR is an OR and MERGE element for events. When an event arrives either of two inputs, there will be an event on the output. When a place corresponding to one of two inputs contains a token, the corresponding transition is enabled, and firing it results in the corresponding place of the output containing a token, i.e. an event will appear on the output of the XOR.

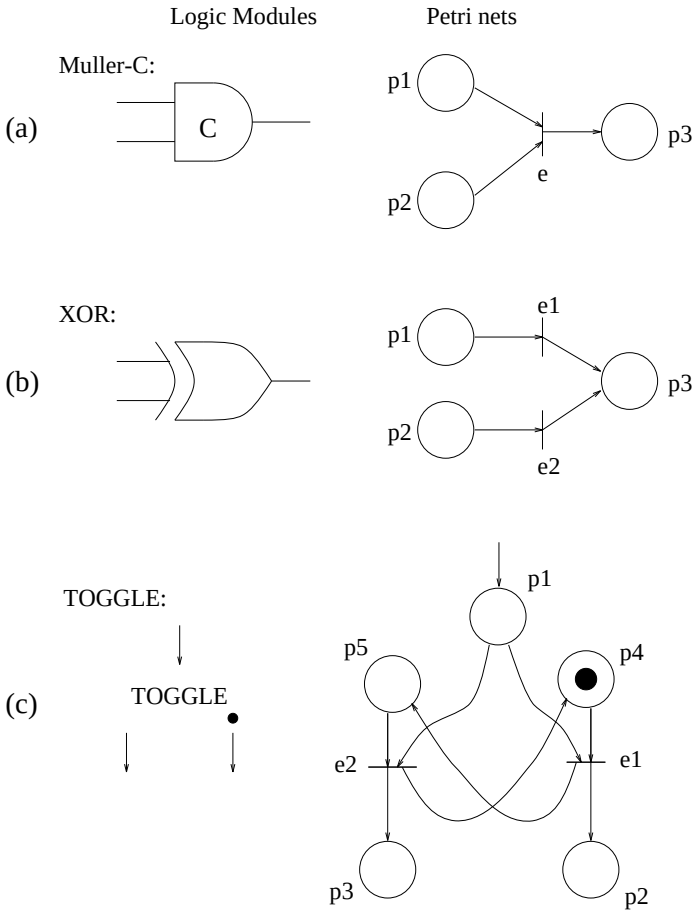


Figure 3.4: Petri net models of a MULLER-C, an XOR and a TOGGLE

TOGGLE module

The Petri net model of a TOGGLE is shown in Figure 3.4(c). A TOGGLE is an ALTERNATE element for events. If the current input is sent to the dotted output, then the next input event will be sent to the non-dotted output. If the current input is sent to the non-dotted output, then the next input event will be sent to the dotted output. This situation will be cyclic for ever. There are two places in the Petri net model of the TOGGLE to control the token denoting events inside the input place and this token will be sent to the dotted output place or the non-dotted output place. Initially the place labelled "p4" contains a token. Therefore, the first input event (denoted by a token inside the input place labelled "p1") will cause the transition labelled "e1" to fire. Then the dotted output place labelled "p2" will contain a token. This means the event has been sent to the dotted output. After this action, the place labelled "p5" contains a token which can cause a subsequent input event to fire the transition labelled "e2", i.e. next time the input event will be sent to the non-dotted output.

SELECT module

The Petri net model of a SELECT module is shown in Figure 3.5(a). A SELECT module checks its Boolean condition and decides whether the incoming event will be sent to the output labelled "true" or the output labelled "false". One important thing is that the Boolean value must arrive before checking it. For this reason, there is a "waiting Boolean" place labelled "p1" in the Petri net model of a SELECT to indicate that the evaluation of producing the Boolean value may be started. If this place contains a token, the transition labelled "e1, execute Boolean" will be fired to start the corresponding Boolean evaluation. Therefore, the correct Boolean value is able to appear on the Bool input of the SELECT module. Then the place labelled "p7, Bool Available" contains a token. Thus,

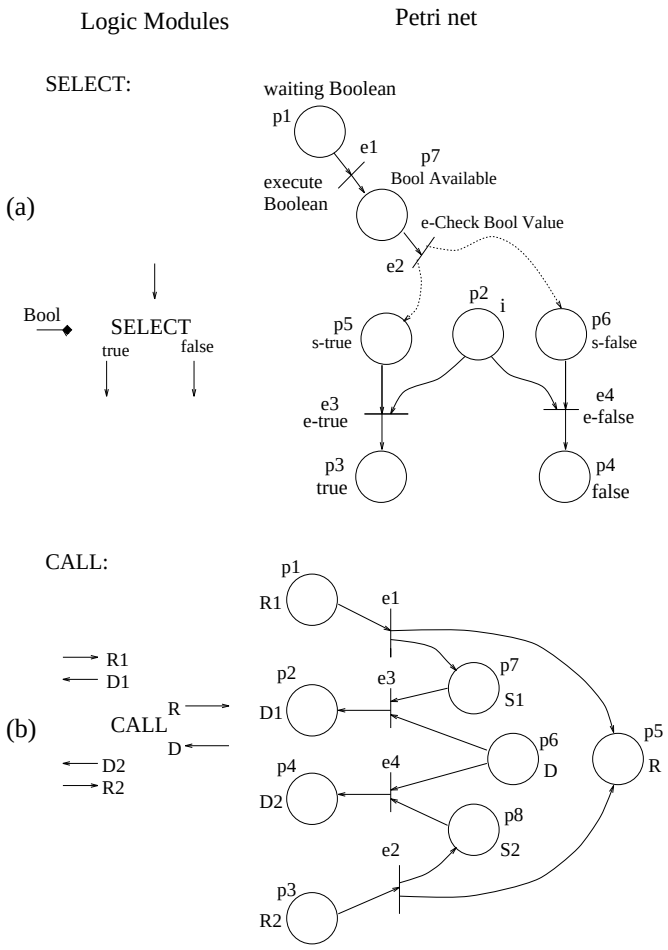


Figure 3.5: Petri net models of a SELECT and a CALL

the transition labelled "e-Check Bool Value" can be fired. Depending whether the Boolean value is true or false, a token is put inside the place labelled "s-true" or the place labelled "s-false". When the incoming event arrives, one of the places labelled "s-true" and "s-false" will fire the transition labelled "e-true" or "e-false". This will cause to the event appear correctly on the place labelled "p3, true" or the place labelled "p4, false".

CALL element

The Petri net model of a CALL element is shown in Figure 3.5(b). A CALL element will remember which of its inputs most recently has received an event, and will return an event to the matching output terminal after the called procedure has finished. There are two places, labelled "p7, s1" and "p8, s2" in Figure 3.5(b) to memorize the input being served. These are used to return the event which denotes that the called procedure has finished to the correct matching output. If an event arrives on the input labelled "R1", there is a token inside the place labelled "R1" to denote this situation. The transition labelled "e1" will fire. Then the places labelled "p7, s1" and "p5, R" are assigned a token each. The place labelled "p5, R" containing a token denotes that there is an event on its output labelled "R". When the called procedure has finished, there is an event on the input labelled "D". The place labelled "p6, D" will contain a token to denote this situation. After firing the transition labelled "e3", the place labelled "p2, D1" will contain a token. If an event arrives on the input labelled "R2", a similar sequence of operations will happen. The difference is that the places labelled "s7, s1", "p2, D1" and the transitions labelled "e1", "e3" will change to the places labelled "s8, s2", "p4, D2" and the transitions labelled "e2", "e4" respectively.

ARBITER module

The Petri net model of an ARBITER is shown in Figure 3.6(a). An ARBITER module decides between two events whose arrival sequence is unknown, producing a grant event for only one of them even if they arrive at very nearly the same time. The token inside the place labelled "p7, S" is used to denote the status of the ARBITER module. It may be "idle" or "busy". If the place contains a token, the ARBITER module is "idle". When an event arrives on one of both inputs, one of the places labelled "p1, R1" and "p4, R4" will contain a token. The token inside the "idle" place can fire the corresponding transition labelled "e1" or "e2". Then the corresponding place labelled "p2, G1" or "p5, G2" will contain a token. This means the event has appeared on the corresponding output. For example, if an event only arrives on the input labelled "R1", i.e. there is a token inside the place labelled "p1, R1", the transition labelled "e1" will fire and a new token will be assigned into the place labelled "p2, G1". When the procedure has finished, there will be an event on the input labelled "D1", i.e. there will be a token inside the place labelled "p3, D1". After the transition labelled "e2" is fired, the "idle" place will contain a token again to treat subsequent events.

When two events arrive very close together on both inputs, the places labelled "p1, R1" and "p4, R4" will both contain a token. Since only one token is inside the "idle" place, only one of the corresponding transitions labelled "e1" and "e3" will fire and the other will be disabled. The output place of the fired transition will be assigned a token. It denotes the event has been sent to the corresponding output. After executing the procedure, there will be a token inside the corresponding place labelled "p3, D1" or "p6, D2" to enable and fire one of the transitions labelled "e2" and "e4", where after the other event will proceed.

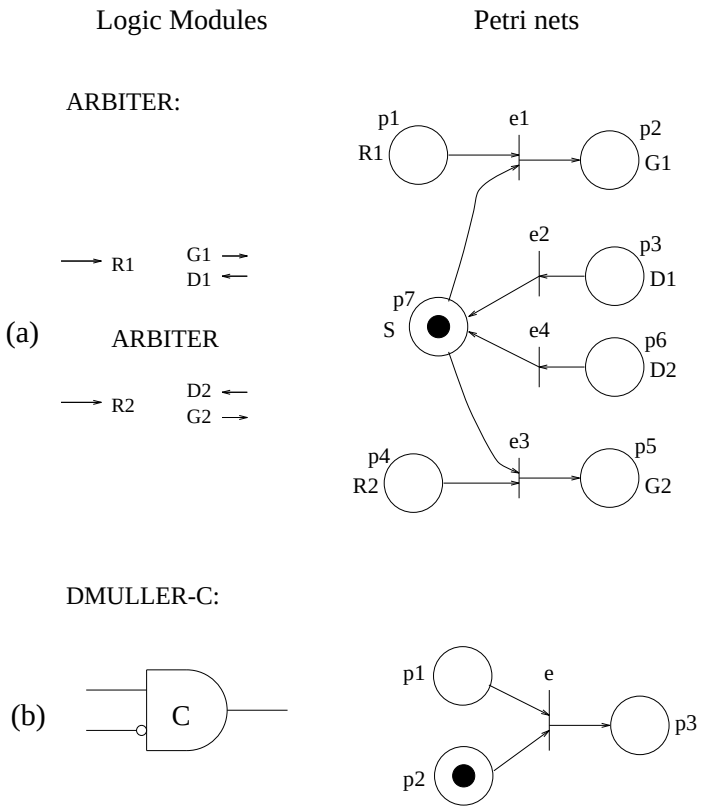


Figure 3.6: Petri net models of an ARBITER and a DMULLER-C

DMULLER C-element

The Petri net model of a DMULLER C-element is shown in Figure 3.6(b). If a DMULLER C-element did not receive an event on its non-inverting input, an event on its non-inverting input can be sent to its output. If a DMULLER C-element has received an event on its non-inverting input and then it has received an event on its inverting input, a subsequent event on its non-inverting input can be sent to its output. If a DMULLER C-element has received an event on its non-inverting input and then it does not receive an event on its inverting input, a subsequent event on its non-inverting input must wait and cannot be sent to its output. Therefore, initially assigning a token inside the place labelled "p2" denotes both inputs have the same state. This token can be used to enable the transition when there is an incoming event on the non-inverting input. Then an event can be sent to the output, i.e., the place labelled "p3" contains a token. However, the following event on the non-inverting input must wait for an event on the inverting input. After an event appears on the inverting input, the transition can be enabled, fired, and cause the event to appear on the output.

Transparent latches

The Petri net models of a low activated latch and a high activated latch are the same. They are shown in Figure 3.7(a) and (b) respectively. A transparent latch will be transparent or holding data, depending on the state of the input labelled "lt". When each event arrives, a transparent latch will change its function to be transparent or holding data. Therefore, a transition and two places, where the one containing a token always consumes the token inside the other when the transition is fired, can be used to model the latch. It is important that if an event arrives on the input labelled "lt", i.e. there is a token inside the place labelled "p1, lt", the transition labelled "e:execute" will fire and the corresponding functions

of the transition which change the action of the latch will be executed.

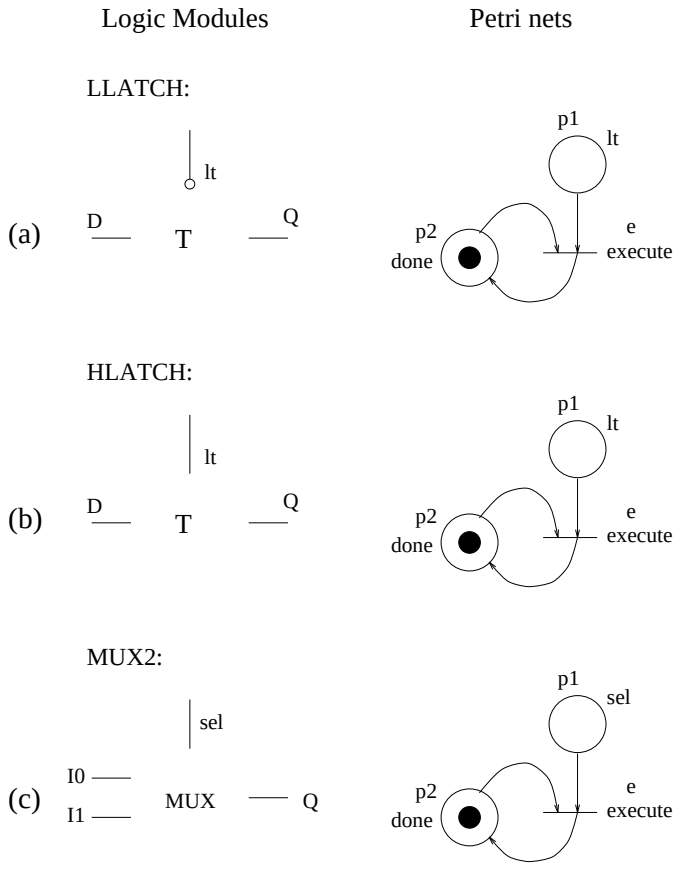


Figure 3.7: Petri net models of Transparent latches and a Multiplexer

Multiplexer

The Petri net model of a Multiplexer is similar to that of a latch. It is shown in Figure 3.7(c). When each event arrives, a multiplexer will change the connection from one of its two input to its output. Therefore, a transition and two places, where the one containing a token always consumes the token inside the other when the transition is fired, can be used to model the multiplexer. It is important that if an event arrives on the input labelled "sel", i.e. there is a token inside

the place labelled "p1, sel", the transition labelled "e:execute" will fire and the corresponding functions of the transition which change the connection from one of two input of the multiplexer to its output will be executed.

TEST PATTERN GENERATOR

The Petri net model of a TEST PATTERN GENERATOR is similar to that of a DMULLER-C. It is shown in Figure 3.8(a). The operating cycle of micropipelines consists of (1) data available, (2) request event, and (3) acknowledge event. Before and after the cycle, the request output and the acknowledge input will have the same state. Initially a token inside the place labelled "p2, ack" denotes that the request output and the acknowledge input have the same state. If there are tokens inside the test pattern queue, a token is put inside the place labelled "p3, Test pattern available". The transition labelled "e:send out" will fires, the test pattern generator reads the front test pattern from the test pattern queue and sends it on the data input of the simulated network. After firing the transition, the output place labelled "p1, req" will be assigned a token denoting an event on the request output of the test pattern generator.

RESULT BUFFER

The Petri net model of an RESULT BUFFER is shown in Figure 3.8(b). It is very simple and just a transition labelled "e:enter data into buffer" with one input place labelled "p1, data waiting" and one output place labelled "p2, entered into queue". If there is some data to be entered into the result buffer, the data must have been available on the input of the result buffer and there must have been an event on the request input of the result buffer. This event can be denoted by a token inside the place labelled "p1, data waiting". Then the transition will be

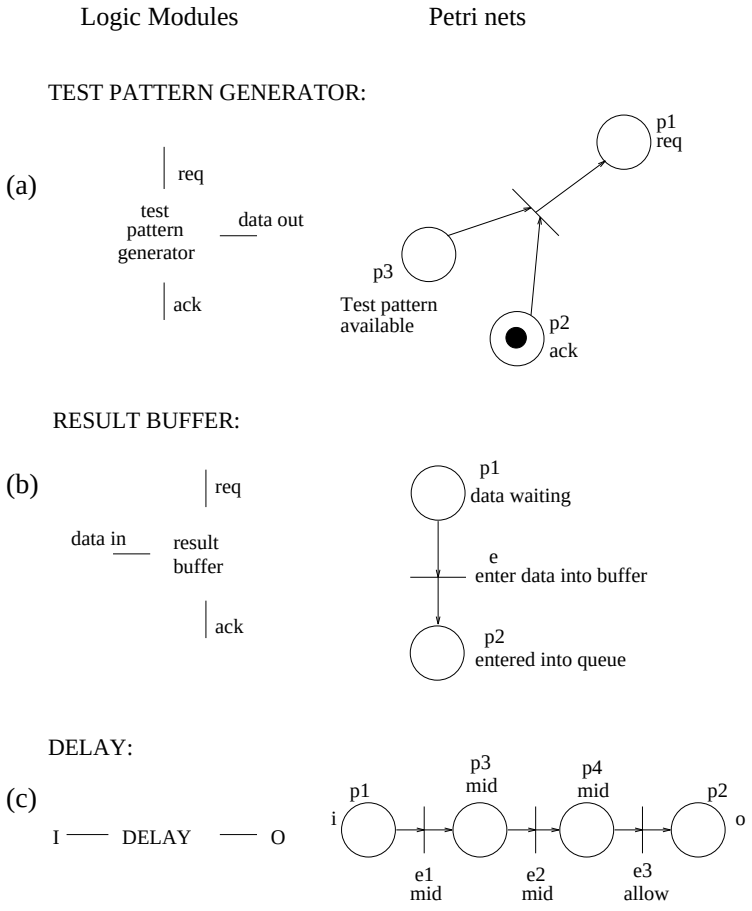


Figure 3.8: Petri net models of a TEST PATTERN GENERATOR, a RESULT BUFFER and a DELAY

enabled and fired to execute the action of entering data into buffer. When this action has finished, the place labelled "p2, entered into queue" will be assigned a token, i.e. an event will appear on the acknowledge output of the result buffer.

DELAY circuit

A DELAY circuit is used to retard the event arriving on the input of event-driven logic modules, which are connected to the output of this DELAY. Its Petri net model is shown in Figure 3.8(c). It consists of four places and three transitions which are connected in series. Among all the petri net models of the event-driven logic modules, there is no numbers of the places and transitions which are connected in series bigger than four and three respectively. Therefore, the token through this Petri net model is not faster than others and the purpose of the delay can be achieved. The four places are labelled "p1, i", "p3, mid", "p4, mid", and "p2, o" respectively. The three transitions are labelled "e1:mid", "e2:mid", and "e3:allow" respectively. If an event is on the input of a DELAY, i.e. there is a token inside the place labelled "p1, i", this event will appear on the output of the DELAY after firing the three transitions. During the firing of these three transitions, the action which needs this delay has finished.

Table 3.1 lists the event functions of various Petri net transitions of the logic modules which are executed when the corresponding transitions are fired.

Table 3.1: The event functions of the Petri net transitions of event-driven modules

The event functions of the Petri net transitions of the logic modules for events		
Logic modules	Transition Name	Function
Muller-C2	muller_c2:e	Check both inputs and change the output state.
Dmuller-C2	dmuller_c2:e	Check both inputs and change the output state.
Mxor2	mxor2:e1	Change the output state.
	mxor2:e2	Change the output state.
Toggle	toggle:e1	Change the dotted output state.
	toggle:e2	Change the nondotted output state.
Select	select:e-execute Boolean	Evaluate the logic device connected to the bool pin of a select. Obtain correct logic value.
	select:e-check Boolean value	Check the logic value on the bool pin: if it is true, the "s-true" place gets a token if it is false, the "s-false" place gets a token.
	select:e-true	Change the output state of the "true" pin.
	select:e-false	Change the output state of the "false" pin.
Call	call:e1	Assign the "S1", "R" places a token and change the output state of the "R" pin.
	call:e2	Assign the "S2", "R" places a token and change the output state of the "R" pin.
	call:e3	Change the output state of the "D1" pin.
	call:e4	Change the output state of the "D2" pin.
Arbiter	arbiter:e1	Change the output state of the "G1" pin.
	arbiter:e2	Return the token to the "s" place.
	arbiter:e3	Change the output state of the "G2" pin.
	arbiter:e4	Return the token to the "s" place.
Mux2	mux2:execute	Consume the token inside the "sel" place.
Llatch1 Hlatch1	llatch1:execute hlatch1:execute	If the value of "lt" pin is 1, it means that the "lt" has gone from 0 to 1. First let lt=0, execute the logic of the input of the low-activated-latches and then evaluate these latches. Then change lt=1, execute the logic of the input of the high-activated latches and then evaluate these latches. If the value of "lt" pin is 0, it means that the "lt" has gone from 1 to 0. First let lt=1, execute the logic of the input of the high-activated-latches and then evaluate these latches. Then change lt=0, execute the logic of the input of the low-activated-latches and then evaluate these latches. Consume the token inside the "lt" place.
Test pattern generator	generator:send out	Send data to the output. Change the output state of the "req" pin.
Output buffer	buffer:enter data into buffer	Get the output data of previous stages and enter them into the queue of the result buffer. Change the output state of the "ack" pin.
Delay	delay:mid	Enable the subsequent transition.
	delay:allow	Send the input event to the output. Change delay time.
Point	point	Enable the subsequent transitions.
Line	line	Send the input state to the output.
Select bool	filter	Check the corresponding latch of the input place holding data or not. If not, remove the token. If all the input places of this transition have a token, enable the subsequent transition.

Chapter 4

Representations of circuits and models

This chapter will define a temporary notation for describing micropipeline circuits and the simulation work for this research stage and present the C++ classes which will be used to represent the circuit model and the Petri net model.

4.1 Definitions for entering circuits and describing the simulation

Chapter 3 has provided Petri net models of event-driven logic models. Before such Petri net models can be used to model micropipelines and then the modelled micropipelines can be simulated, consideration must be given to how micropipeline circuits can be entered and how the simulation work of the modelled micropipelines can be specified. This section will define two sets of descriptions. One will be used to enter micropipeline circuits. The other will be used to specify the simulation work, i.e. how the test patterns are entered into the modelled micropipeline circuits and then how the test results are read from the modelled micropipeline circuits. These two kinds of definitions will be described in the next subsections.

4.1.1 Definitions for entering circuits

In this subsection a simple method for entering the simulated circuits will be defined. Its format and definitions are shown as follows:

1. Format:

```
(1) {stage} {:} {space} {stagename} {,}

(2) {device function keyword} {:} {space} {pointname} {,}
                                  [{space} {pointname} {,}]*

(3) {network} {:} {space} {networkname} {,}
```

stage: identifies the start of a new micropipeline stage.

device function keyword: denotes logic functions using a group of abbreviations.

For example: a two-input AND gate is denoted by "and2".
a three-input OR gate is denoted by "or3".

network: identifies the start of the inter-stage network description.

pointname: represents input and output terminals to a device.

The "device function keyword"s of the related logic devices are as follows:

(a) mux2: denotes a 2:1 multiplexer.
(b) ltlatch1: denotes a one-bit "low" activated latch.
(c) htlatch1: denotes a one-bit "high" activated latch.
(d) and2: denotes a two-input AND gate.
(e) fulladder1: denotes a one-bit full adder.
(f) nor3: denotes a three-input NOR gate.
(g) muller-c2: denotes a two-input Muller C-element.
(h) nmuller-c2: denotes a Muller C-element with two inputs and one inverted output.
(i) dmuller-c2: denotes a Muller C-element with one inverted input and one non-inverted input.

(j) mxor2: denotes a two-input XOR gate.

(k) xor2: denotes a two-input XOR gate.

(l) toggle: denotes a "TOGGLE" logic module for events.

(m) select: denotes a "SELECT" logic module for events.

(n) call: denotes a "CALL" logic module for events.

(o) arbiter: denotes an "ARBITER" logic module for events.

(p) or2: denotes a two-input OR gate.

(q) not: denotes a NOT gate.

(r) delay: denotes a delay device for some time.

(s) rin: denotes a Request for inputs. (from previous stages)

(t) ain: denotes an Acknowledge for inputs.

(u) rout: denotes a Request for outputs. (to next stages)

(v) aout: denotes an Acknowledge for outputs.

(w) input: denotes a data input.

(x) output: denotes a data output.

2. Examples:

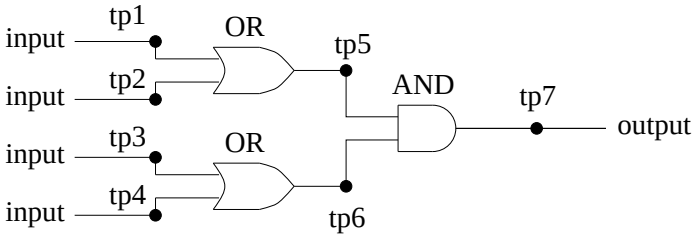


Figure 4.1: The circuit example used to illustrate a circuit description.

(a) The circuit of Figure 4.1 can be described as follows:

```
or2: tp1, tp2, tp5,
or2: tp3, tp4, tp6,
and2: tp5, tp6, tp7,
input: tp1,
input: tp2,
input: tp3,
input: tp4,
output: tp7,
```

(b) The micropipeline stage of Figure 2.4 can be described as follows:

```
stage: latch,
  ltlatch1: lt, c, y,
  or2: a1, a2, a,
  or2: b1, b2, b,
  and2: a, b, c,
  dmuller-c2: ri, w, dmy1,
  mxor2: dmy1, ao, lt,
  toggle: lt, ai, w,
  rin: ri,
  ain: ai,
  rout: dmy1,
  aout: ao,
  input: a1,
  input: a2,
  input: b1,
  input: b2,
  output: y,
network: project,
```

4.1.2 Simulation description

The simulator needs an input file for describing how the test pattern generators and output buffers are connected to the simulated network. The following formats can be used to write Simulation descriptions and test patterns. They are saved into an input file.

1. Format:

```
(1) {command} {:} {space} {pointname} {,}
                  [{space} {pointname} {,}]*

(2) {xv} {:} {space} {test pattern} {,}
             [{space} {test pattern} {,}]*
             {expected test result} {,}
             [{expected test result} {,}]*
```

command: defines which "Rin", "Ain", "Rout", "Aout" are connected to the test pattern generator and the result buffer.

xv: sets test pattern vectors and expected test results.

Commands:

- defrin: defines the Rin of the simulated network.
- defain: defines the Ain of the simulated network.
- definput: defines the Din of the simulated network.
- defrout: defines the Rout of the simulated network.
- defaout: defines the Aout of the simulated network.
- defoutput: defines the Dout of the simulated network.
- defformat: defines the order of the test pattern vectors and the expected test results.
- deftest: starts to set test pattern vectors and expected test results.
- endtest: indicates the end of the simulation.

2. Examples: the following simulation description can be used to simulate the circuit shown in Figure 2.4. The last digit in each test pattern vector is the expected output at the point labelled "y" when a set of "a1", "a2", "b1" and "b2" is entered into the simulated network.

```
defrin: latch#ri,
defain: latch#ai,
definput: latch#a1, latch#a2, latch#b1, latch#b2,
defrout: latch#dmy1,
defaout: latch#ao,
defoutput: latch#y,
defformat: latch#a1, latch#a2, latch#b1, latch#b2, latch#y,
deftest:
xv: 0 0 0 0      0
xv: 0 0 0 1      0
xv: 0 0 1 0      0
xv: 0 0 1 1      0
xv: 0 1 0 0      0
xv: 0 1 0 1      1
xv: 0 1 1 0      1
xv: 0 1 1 1      1
xv: 1 0 0 0      0
xv: 1 0 0 1      1
xv: 1 0 1 0      1
xv: 1 0 1 1      1
xv: 1 1 0 0      0
xv: 1 1 0 1      1
xv: 1 1 1 0      1
xv: 1 1 1 1      1
endtest:
```

4.2 The C++ language

The C++ programming language [Lippman 90] [Stroustrup 91] [Ellis 90] was developed at AT&T Bell Laboratories in the early 1980's by Bjarne Stroustrup. It provides users with a programming environment of data abstractions and object-oriented design. It is an extension of the C language. C++ is a strongly typed language. All initializations and assignments of values and both the argument list and the return type of every function call will be checked at compile time to make sure the types of these values and function call are correctly matched. C++ provides pointer types which can be used to create linked lists. C++ has some useful properties, such as user-defined data types, classes, function name overloading and free storage. To model the micropipeline circuit models and Petri net models of each event-driven logic module, each data item within such models, such as devices, terminals, places, and transitions, must be represented by a data type. The abstract class data type of C++ is easily able to represent these data items. The free storage feature of C++ conveniently supports the management such of data items. The object-oriented nature of C++ supports potential future development of this modelling research. This is why C++ was chosen to be the modelling tool for this research work. These useful features will briefly be described in the following subsections.

4.2.1 Function name overloading

Function name overloading allows multiple function instances that provide a common operation on different argument types to share a common name. Overloading allows a set of functions that perform a similar operation to be collected under a common mnemonic name. In C++, two or more functions can be given the same name provided that their signatures are unique, in either the number or the types of their arguments. Function name overloading relieves the problem of

lexical complexity and makes the program more readable.

4.2.2 Free storage

Storage allocation that occurs at compile time is referred to as static memory allocation. The allocation of memory at run time is referred to as dynamic memory allocation. C++ free storage allows a program to do memory allocation at run time. A *new* operator handles dynamic memory allocation. Free store memory is allocated by applying the operator *new* to a type specifier, including that of a class name. Either a single object or an array of objects can be allocated. Deallocation is achieved by applying the operator *delete* to a pointer addressing the dynamic object. The numbers of devices and terminals within various micropipeline circuits are different. The numbers of places and transitions within the corresponding Petri net models of various micropipeline circuits are also different. Therefore, variable size lists are used to manage these data items. A linked list with variable size is implemented by using this property of dynamic allocation and the pointer types. Any required memory locations can be newly created using the *new* operator for any data items at run time when such data items are met. It is also used to store the input data whose size is not known in advance.

4.2.3 Classes

In C++, a class is a user-defined data type which is an aggregate of named data elements, possibly of different types, and a set of operations designed to manipulate that data. Typically, a class is used to introduce a new data type into the program. A well-designed class can be used as easily as a predefined data type. A C++ class has four associated attributes: data members, member functions, specifications of the levels of program access, such as *private*, *protected*,

or *public*, and a class tag name. A class definition consists of two parts: the class head, composed of the keyword class and tag name, and the class body, enclosed by braces and terminated with a semicolon. A class tag name represents a new data type. Within the class body, the data members, member functions, and levels of information hiding are specified. Information hiding is a formal mechanism for restricting user access to the internal representation of a class type. In general, only the member functions of a class are permitted to access the class data members. If the data representation of the class is changed, only the class member functions, rather than user programs, need to be modified. If there is an error in the manipulation of a class data member, only the small set of class member functions, rather than an entire program, need to be examined.

Members declared within a public section become public members; those declared within a private or protected section become private or protected members. A public member is accessible from anywhere within a program. A protected member behaves as a public member to a derived class; it behaves as a private member to the rest of the program. A private member can be accessed only by the member functions and friends of its class.

A member function is designated the class destructor by giving it the tag name of the class prefixed with a tilde "$\sim$". It is the deallocation function and is invoked whenever an object of its class goes out of scope or the operator *delete* is applied to a class pointer.

An initialization member function, called a constructor, is implicitly invoked each time a class object is defined or allocated by the operator *new*. A constructor is specified by giving it the class name.

A class object or reference accesses its data members or member functions using the class object selector "$\cdot$". A pointer to a class object accesses data members or member functions of the pointed class object using the class pointer

selector "$->$". One important use of the C++ class mechanism is referred to as an abstract data type. Another important use of the class mechanism is to define subtype relationships.

Each device of a circuit has some input and output terminals. Each terminal of a circuit may be connected to several devices. Similarly, each place of a Petri net is connected to several transitions and each transition of a Petri net is connected to several places. Such elements within a circuit and a Petri net can easily be represented by the objects of various classes. The implementation of communications between these elements can be simplified by this data type.

4.2.4 Object-oriented programming

Inheritance and dynamic binding are two primary features of object-oriented programming. Inheritance is implemented through the mechanism of class derivation. An inheritance hierarchy defines a type/subtype relationship between class types. Class inheritance allows the members of one class to be used as if they were members of a second class. Dynamic binding is provided by virtual class functions. A member function is made virtual by preceding its declaration in the class body with the keyword *virtual*. This technique is not applied at this design stage. However it is being considered for future use to reduce the linkages between objects which are used to represent the elements within circuits and Petri nets at subsequent design stages.

4.3 The Design of C++ classes for representing simulated circuits and Petri nets

To model event-driven logic modules and implement a simulator, some classes have been defined. Their global functions, member functions, and data members are shown in Table 4.1. A brief introduction is as follows:

Device and Point class The circuit within a stage consists of devices and terminals. The Device class is defined for denoting the device elements. The Point class is defined for denoting the terminal elements. The data member labelled "name" within each Device object is used to represent the logic function of the represented device. The data member labelled "val" within each Point object is used to represent the logic value which the terminal has. Each Device object has some pointers pointing to Point objects which represent the terminals connected to this device object. Similarly, each Point object also has some pointers pointing to Device objects which represent the devices connected to this point object. To represent different devices, each Device object also has some other data members to denote the data which the particular device needs. For example, each Device object has a pointer to point to the Place objects and Event (i.e. transition) objects in the corresponding Petri net model if the represented device is an event-driven module. Each Device object has a queue to store the test patterns or the test results if the represented device is a test pattern generator or a result buffer. To access conveniently these two kinds of objects, the DeviceList class and the PointList class are also defined. All the Device objects and all the Point objects within a stage will be stored as linked lists individually. The related global functions and member functions are used to manage their data members.

Table 4.1: The global functions, member functions, and data members of C++ classes for modelling the Petri net models of logic modules for events

	class name					
	Device	Point	Stage	Network	Place	Event
Global Function	setPoint getPoint changePoint display setPlace getPlace changePlace setEvent getEvent changeEvent	setDevice getDevice			setPreEvent getPreEvent changePreEvent setPostEvent getPostEvent changePostEvent display addPlaceList remove_from_PlaceList	setPrePlace getPrePlace changePrePlace setPostPlace getPostPlace changePostPlace display addEventList remove_from_EventList
Member Function	Device disp setpoicnt getpoicnt setplaceCount getplaceCount seteventCount geteventCount setpinPtr setbotpinPtr getpinPtr getbotpinPtr setplacePtr setbotplacePtr getplacePtr getbotplacePtr seteventPtr setboteventPtr geteventPtr getboteventPtr setState getState changeState insertQ removeQ getQCount getstagename setstagename setdelayTime getdelayTime	Point set get disp getdevcnt setdevcnt setdevicePtr setbotdevicePtr getdevicePtr getbotdevicePtr insertTimeQ removeTimeQ getrearTimeQ getTimeQCount	Stage disp display setTopDevPtr setTopPoiPtr getTopDevPtr getTopPoiPtr getDeviceCount getPointCount	Network display disp setTopStgPtr setTopDevPtr setTopPoiPtr getTopStgPtr getTopDevPtr getTopPoiPtr getStageCount getDeviceCount getPointCount	Place disp setToken getToken removeToken setDevice getDevice setPreEventPtr setbotPreEventPtr getPreEventPtr getbotPreEventPtr setPostEventPtr setbotPostEventPtr getPostEventPtr getbotPostEventPtr setPreEventCount setPostEventCount getPreEventCount getPostEventCount	Event disp setDevice getDevice getDeviceCount setPrePlacePtr setbotPrePlacePtr getPrePlacePtr getbotPrePlacePtr setPostPlacePtr setbotPostPlacePtr getPostPlacePtr getbotPostPlacePtr setPrePlaceCount setPostPlaceCount getPrePlaceCount getPostPlaceCount
Data Member	name stagename poicnt placeCount eventCount stateCount Qcount delayTime pinPtr botpinPtr placePtr botplacePtr eventPtr boteventPtr statePtr botstatePtr frontQ rearQ	pointName val devcnt TimeQCount devicePtr botdevicePtr frontTimeQ rearTimeQ	stageName TopDevPtr TopPoiPtr deviceCount pointCount	networkName TopStagePtr TpoDevPtr TopPoiPtr StageCount deviceCount pointCount	placeName Token deviceCount devicePtr botdevicePtr preEventPtr botpreEventPtr postEventPtr botpostEventptr preEventCount postEventCount	eventName deviceCount devicePtr botdevicePtr prePlacePtr botprePlacePtr postPlacePtr botpostPlaceptr prePlaceCount postPlaceCount

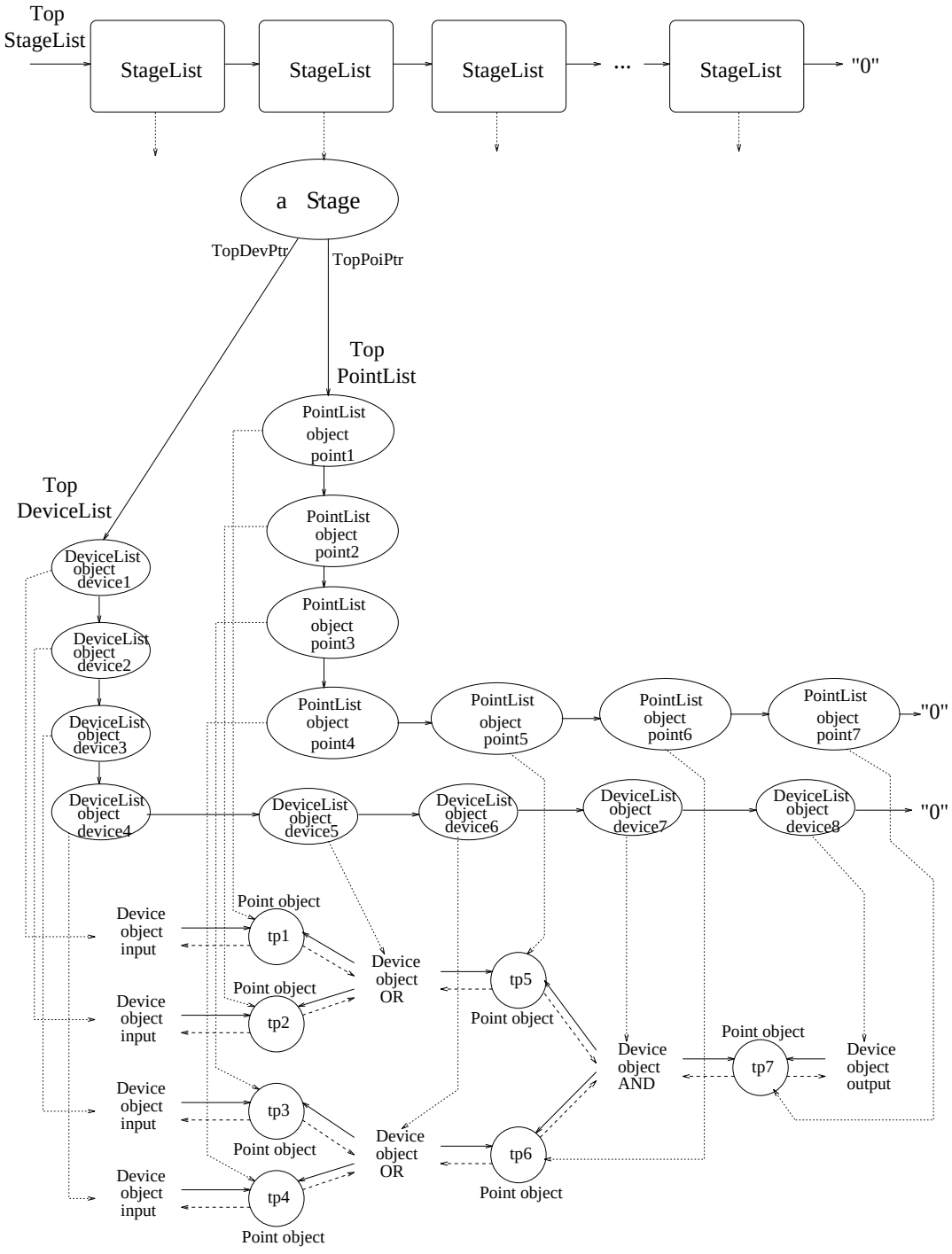


Figure 4.2: Describing circuits within a stage using the Stage object

Stage class The circuits within a stage are stored in a Stage object. There are two important data members in each Stage object. One is the TopDevPtr, a pointer pointing to the top position of the device list which contains all Device objects within this stage. The other is the TopPoiPtr, a pointer pointing to the top position of the point list which contains all Point objects within this stage. Figure 4.2 shows how the stage of Figure 4.1 is denoted by the related C++ classes. The StageList class is necessary in order to access Stage objects conveniently. All Stage objects are stored in a list. TopStageList is the top position of the stage object list.

Network class A pipeline network consists of several stages and some device elements between stages. In a Network object, in addition to the pointer, TopStageList, it is also necessary to point to a device list which contains the devices between stages and a point list in which the terminals connect to such devices. Therefore, there are another two pointers, i.e. the TopDeviceList and the TopPointList, in a Network object pointing to the top positions of the device list and the point list. Figure 4.3 shows the structure of a Network object.

Place class and Event class The objects of these two kinds of classes are used to represent the places and the transitions in a Petri net respectively. Each Place object has two pointers which point to transition lists containing the predecessor and successor transitions. Each Place object has a pointer to a device list which contains all the devices pointing to this place. Similarly, each Event object has two pointers which point to place lists containing the predecessor and successor places. Each Event object has a pointer to a device list which contains all the devices pointing to this transition. Some global and member functions of the Device, Place, and Event classes are used to handle the above pointers to establish the correct mapping relations.

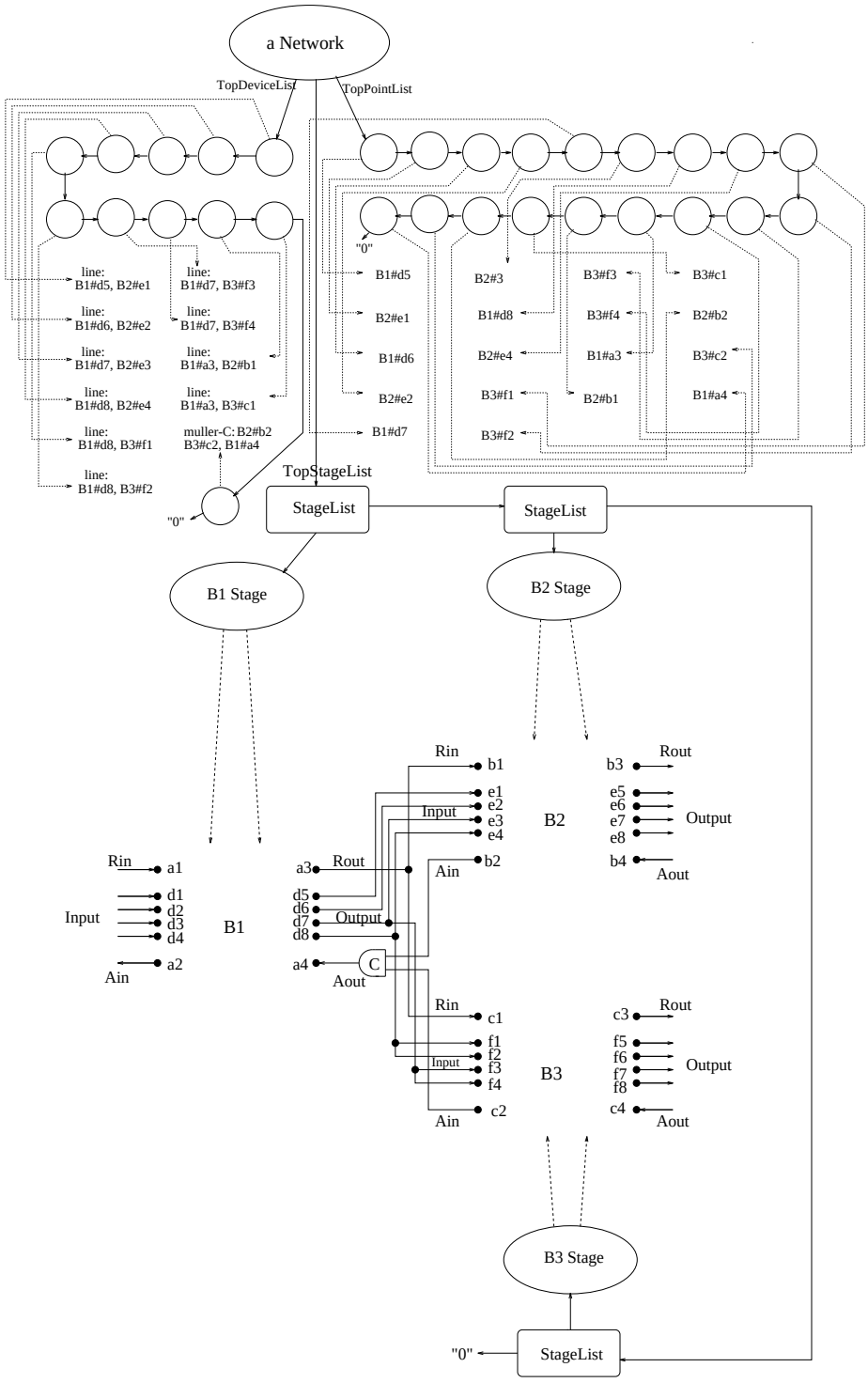


Figure 4.3: The structure of a Network object

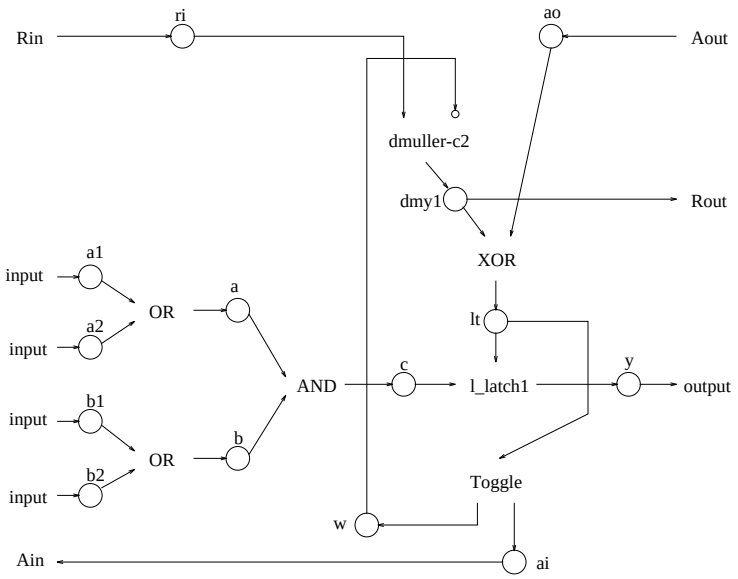


Figure 4.4: The circuit model represented by C++ classes

4.3.1 The representation of circuit models

The micropipeline circuit shown in Figure 2.4 can be represented by the C++ classes described in the previous section as shown in Figure 4.4. The rectangle boxes denote devices, i.e. Device objects. The circles denote terminals, i.e. Point objects. The arrows denote the interconnections between devices and terminals. The arrow directions denote the input and output relations.

4.3.2 The construction of Petri net models

The Petri net model can be constructed using the global functions and member functions of the C++ classes shown in Table 4.1. Some functions are used to establish the mapping from places to transitions or from transitions to places. Some functions are used to establish the mapping from Petri net models to device models. For example, the logic module TOGGLE can be modelled as shown in

Figure 4.5. The first step is to create five Place objects which are pointed to by five Place pointer variables **p1**, **p2**, **p3**, **p4**, and **p5** (See the Petri net model of the TOGGLE module). The second step is to create two Event objects which are pointed to by two Event pointer variables **e1**, **e2** and used to denote the transitions of the Petri net model of the TOGGLE module. The third step is to establish the mapping from Petri net places to the device, from Petri net transitions to the device, from the device to Petri net places, from the device to Petri net transitions, from Petri net transitions to Petri net places, and from Petri net places to Petri net transitions. The fourth step is to assign initially one token into the place pointed to by variable **p4**. The last step is to add these Place objects and Event objects to the place list and the event list respectively.

```
// The implementation of the Petri net model of a TOGGLE
void Petri_toggle(Device* d)  {

// variable declaration
Place *p1, *p2, *p3, *p4, *p5;     Event *e1, *e2;

// step 1: new places declaration
p1 = new Place("toggle:i");        p2 = new Place("toggle:dot");
p3 = new Place("toggle:non-dot");  p4 = new Place("toggle:s1");
p5 = new Place("toggle:s2");

// step 2: new events declaration
e1 = new Event("toggle:e1");       e2 = new Event("toggle:e2");

// step 3: establish mapping device from Petri net places
p1->setDevice(d);   p2->setDevice(d);   p3->setDevice(d);
p4->setDevice(d);   p5->setDevice(d);
e1->setDevice(d);   e2->setDevice(d);

// establish mapping Petri net places from a device
setPlace(d, p1);    setPlace(d, p2);    setPlace(d, p3);
setPlace(d, p4);    setPlace(d, p5);

// establish mapping Petri net transitions from a device
setEvent(d, e1);    setEvent(d, e2);

// establish mapping Petri net transitions from Petri net places
setPostEvent(p1, e1); setPostEvent(p1, e2); setPreEvent( p2, e1);
setPreEvent( p3, e2); setPreEvent( p4, e2); setPostEvent(p4, e1);
setPreEvent( p5, e1); setPostEvent(p5, e2);

// establish mapping Petri net places from Petri net transitions
setPrePlace( e1, p1); setPrePlace( e1, p4); setPostPlace(e1, p2);
setPostPlace(e1, p5); setPrePlace( e2, p1); setPrePlace( e2, p5);
setPostPlace(e2, p3); setPostPlace(e2, p4);

p4->setToken();  // step 4: initial token

// step 5: add places and transitions to PlaceList and EventList
addPlaceList(p1);  addPlaceList(p2);  addPlaceList(p3);
addPlaceList(p4);  addPlaceList(p5);
addEventList(e1);  addEventList(e2);  }
```

Figure 4.5: Modelling a TOGGLE module using C++ classes

Chapter 5

Implementing a micropipeline simulator

This chapter will present the design and operation of the micropipeline simulator. The construction of the corresponding Petri net model of the simulated network will be described. The simulation procedure which will be used to control the simulation work will also be presented in this chapter. Several micropipeline examples will be tested to demonstrate that the simulator works correctly.

5.1 Introduction

The representation of a circuit model using C++ classes and the method for entering a simulated network using the description language have been described in Chapter 4. The Petri net models of event-driven logic modules have also been described in Chapter 3. This chapter will present the micropipeline simulator which is implemented using C++ classes and show how Petri net models of the behaviours of event-driven logic modules are used to control the simulation execution. The simulator design can be divided into three sections: user interface, model description and simulation control. The user will input the logic devices and networks for simulating via the user interface. The user interface can also be

used to input the simulation commands and monitor the progress of the simulation. The definitions for entering circuits and describing the simulation described in Chapter 4 are used to support this input function.

The model description is an internal representation of the design which is suitable for use by the simulator. It is necessary to consider the input pins, the output pins, and the relationship between them, i.e. the truth table. The class structure of the C++ language can be used to represent each device and the pointers to link one device to others within a pipeline stage. The pointers can also be used to link one stage to other stages.

The simulation control is an execution sequence. When the simulation is executed, the test pattern data can flow through a network based on the organisation of connections in the network. When execution terminates, the results will be saved in a file, which can be compared with expected values to observe whether the simulated network is correct or not. Also the values of the state change times of each terminal can be recorded and saved into a file during the simulation. Then they can be displayed as waveforms. A simulation procedure is designed to schedule the Petri net transitions which are enabled to generate the control sequences. After the simulator reads the simulation network described by the user interface, it constructs not only a circuit model but also a Petri net model. Similarly, we can use the classes and pointers of C++ to build this Petri net model during the simulation.

The top level flow chart of the micropipeline simulator is shown in Figure 5.1. It denotes that the simulation work can be divided into five parts: reading the simulated network, constructing the corresponding Petri net model of the network, reading the test patterns, executing the simulation action of the network and displaying the test results of the simulation. Each of these stages will be introduced in the following subsections.

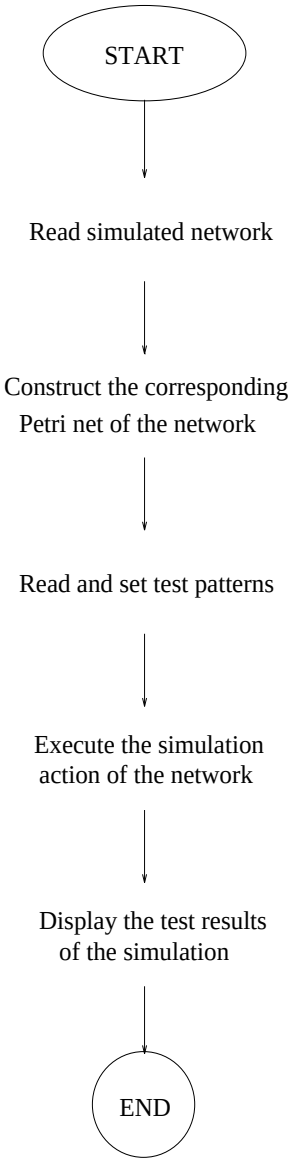


Figure 5.1: The top level flow chart of the micropipeline simulator

5.2 Reading the circuit and simulation descriptions

The program whose flow chart is shown in Figure 5.2 can read circuit and simulation descriptions from an input file and construct a network model using the objects of the Device class and the Point class. Figure 5.3 shows a simulated network model, which is the micropipeline circuit shown in Figure 2.4, and the descriptions described in Chapter 4 are executed by the above program. A test pattern generator has been connected to the input end of the simulated network, and a result buffer to the output end.

5.3 Constructing the corresponding Petri net model of the simulated network model

The construction of the corresponding Petri net model of the simulated network can be divided into three steps. The first step is to produce the corresponding Petri net models of each event-driven logic module within each stage of the simulated network. The corresponding Petri net models of each event-driven logic module have been described in Chapter 3. The second step is to merge the corresponding Petri net models of each event-driven logic module within each stage of the simulated network along each point within the simulated network. Finally, merge the corresponding Petri net models of each stage within the simulated network along with each event-driven logic module between stages.

The program whose flow chart is shown in Figure 5.4 is used to construct the corresponding Petri net model of the simulated network. Let us illustrate this with an example to show how it works. When the corresponding Petri net model of the circuit shown in Figure 2.4 is constructed, the result of the first step is

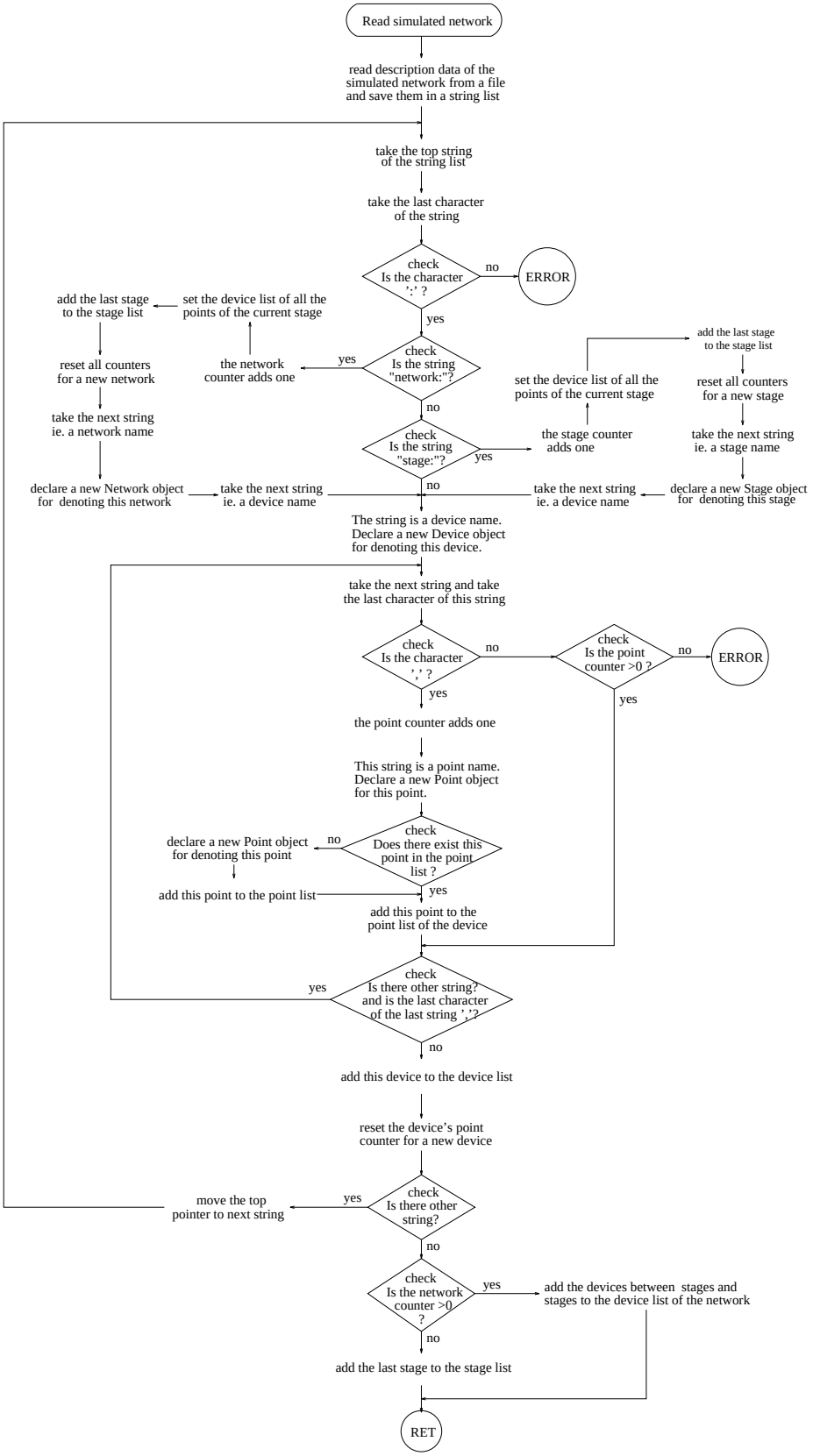


Figure 5.2: The flow chart of a function for reading circuit and simulation descriptions

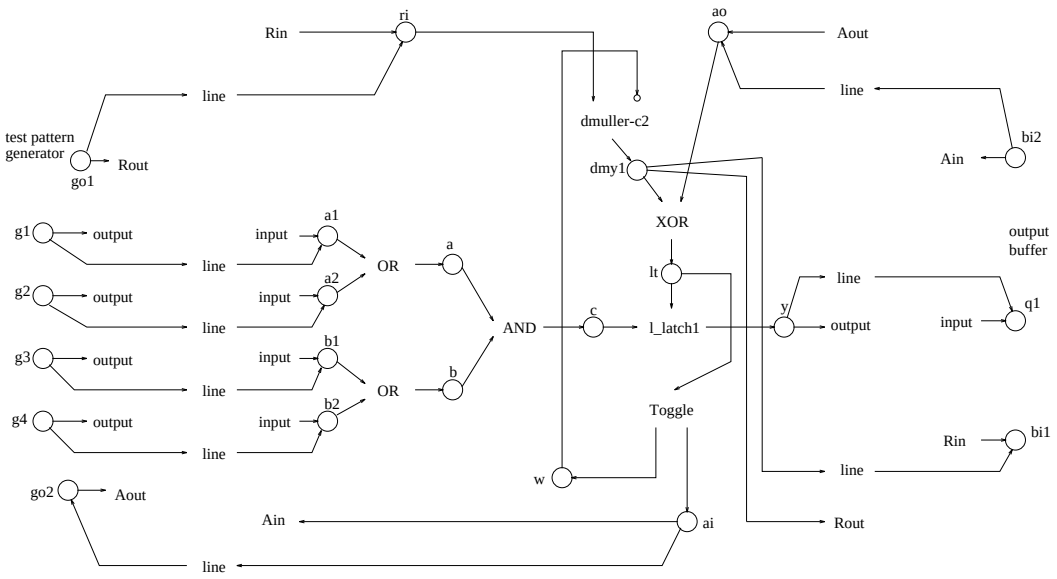


Figure 5.3: The simulated network with the test pattern generator and the result buffer

shown in Figure 5.5. A Petri net model of each of the event-driven modules within each stage will be produced. Next the simulator progresses to the second step, and the corresponding Petri net model of each stage of the simulated network is constructed as shown in Figure 5.6. As the last step, the corresponding Petri net model of the simulated network is produced. Figure 5.7 shows this corresponding Petri net model.

The simulator controls the tokens which denote events flowing through the Petri net model to complete the simulation execution. The next issue is how the test patterns are put into the test pattern generator.

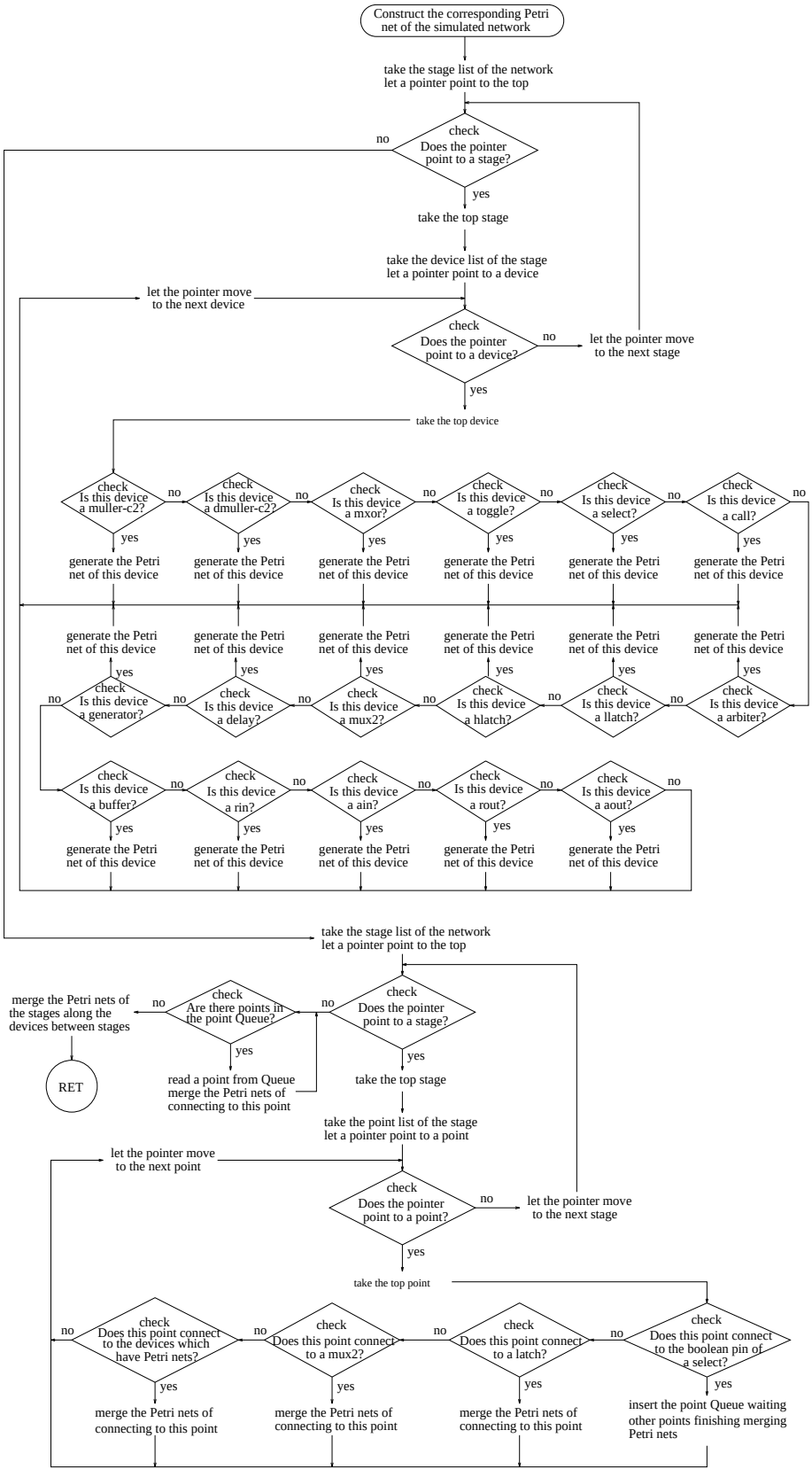


Figure 5.4: The flow chart of a function for constructing the corresponding Petri net model of the simulated network

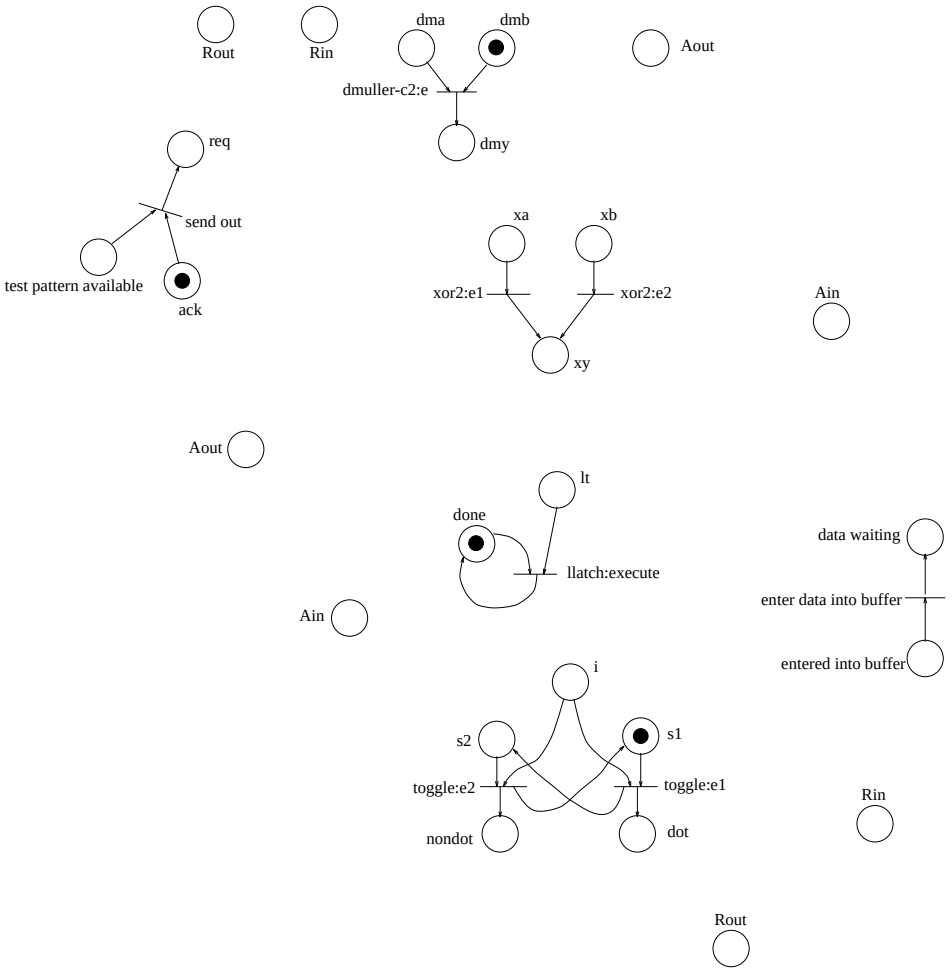


Figure 5.5: Construct the corresponding Petri net model of the simulated network (step 1 : Produce the Petri net model of logic modules for events within stages)

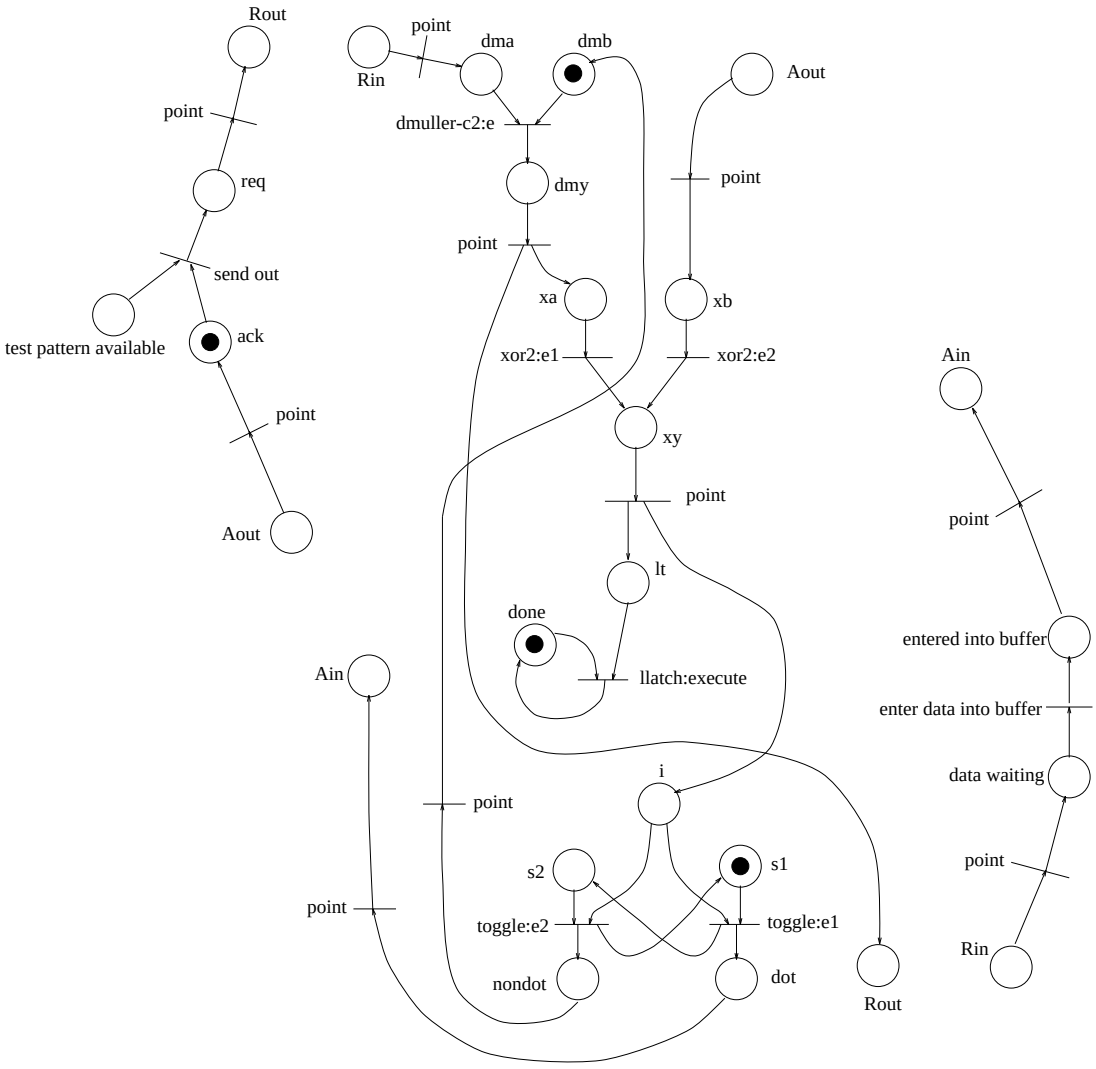


Figure 5.6: Construct the corresponding Petri net model of the simulated network (step 2 : Connect the Petri net models within each stage along each point)

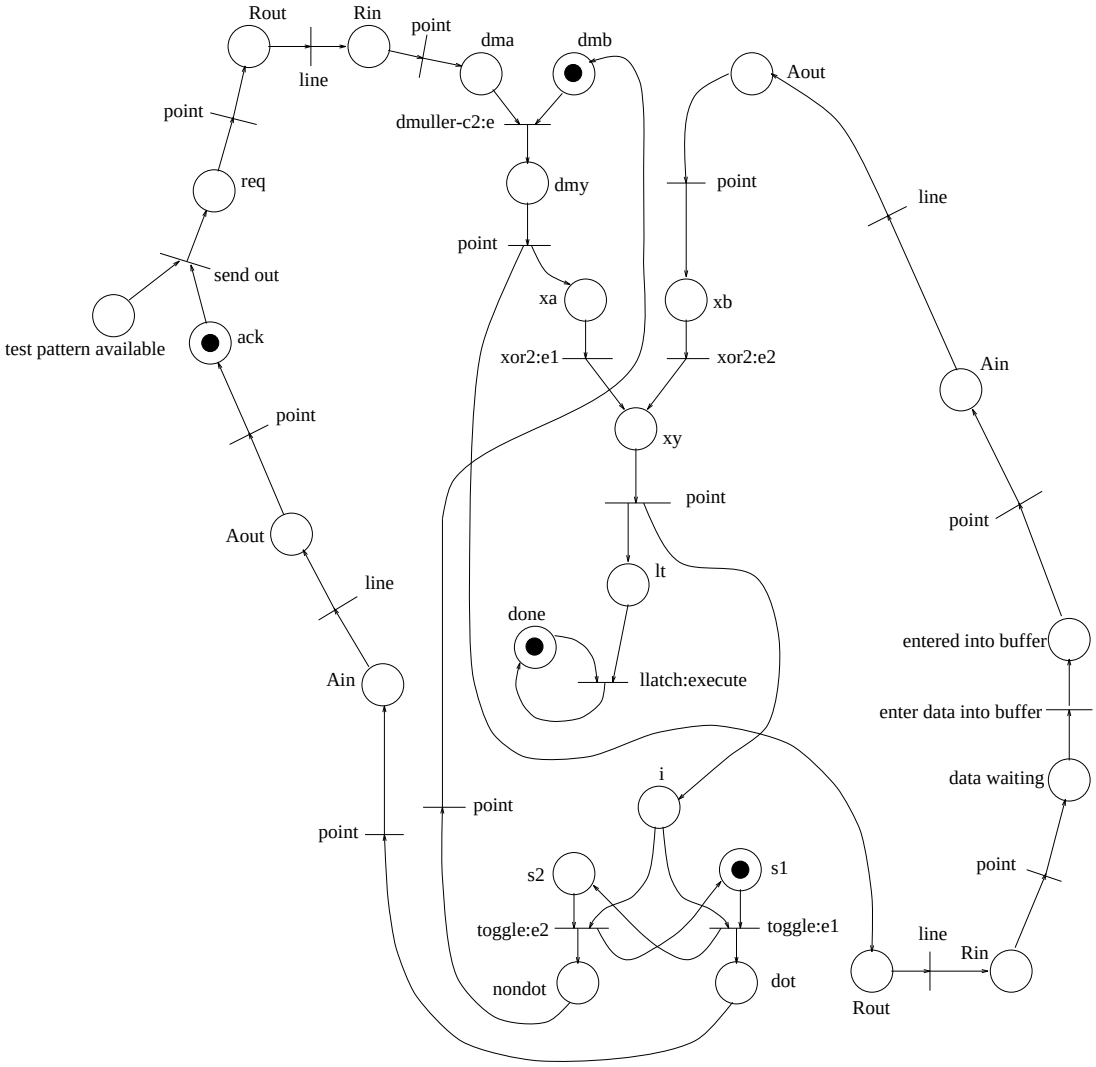


Figure 5.7: Construct the corresponding Petri net model of the simulated network (step 3 : Connect the Petri net models of each stage along each connection device between stages)

5.4 Test pattern input

Before the simulator starts the simulation work, it needs to read the test patterns from an input file. This work is done by the program whose flow chart is shown in Figure 5.8. It is very simple in that the simulator simply reads the test patterns from an input file and saves them into an array, and then follows the order specified by the simulation description command, "defformat" to insert these test patterns into the test pattern queue of the corresponding test pattern generator from that array. After finishing this test pattern input, the simulation work can be executed under the simulation procedure.

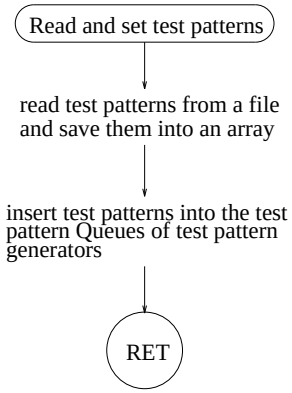


Figure 5.8: The flow chart of a function for reading and setting test patterns

5.5 Network simulation

During the simulation work, the simulator needs a waiting fire queue in order to dispatch concurrently the transitions of the Petri net model which are enabled. The simulation procedure which can be used to control the simulation work concurrently is as follows:

Simulation procedure:

1. Check whether the test pattern generators have test patterns or not.

   (a) If there are some test patterns, then

   Check whether the place labelled "test pattern available" in the Petri net model has a token or not.

   i. If there is a token inside the place, then

   Go to next step.

   ii. If there is no token inside the place, then

   Put a token inside the place and

   Check whether the transitions pointed to by this place are enabled or not.

   A. If there are some transitions which are enabled, then

   Insert the enabled transitions into the waiting fire queue.

   B. If there is no transition which can be enabled, then

   Go to next step.

   (b) If there is no test pattern, then

   Go to next step.

2. Check whether the waiting fire queue contains transitions or not.

   (a) If there are some transitions in the waiting fire queue, then

   Read the front one and fire it(if it is disabled, go to next step.),

   Execute the actions of the devices,

   Remove a token from each predecessor place and

   Add a token into each successor place which is pointed by this fired transition.

Check whether the transitions pointed by each successor place are enabled or not.

i. If there are some transitions which can be enabled, then
Insert the enabled transitions into the waiting fire queue.

ii. If there is no transition which can be enabled, then
Go to next step.

(b) If there is no transition in the waiting fire queue, then
Go to next step.

3. Check whether there is no transition which can be fired and whether the test pattern generators still have some test patterns but the simulator is not able to put a token inside the place labelled "test pattern available" (check deadlock).

(a) If a deadlock has happened, then
Abort the simulation execution.

(b) If no deadlock happens, then
Repeatedly execute the above steps until there is no test pattern and no enabled transitions in the waiting fire queue.

The simulation procedure is implemented by the program whose flow chart is shown in Figure 5.9.

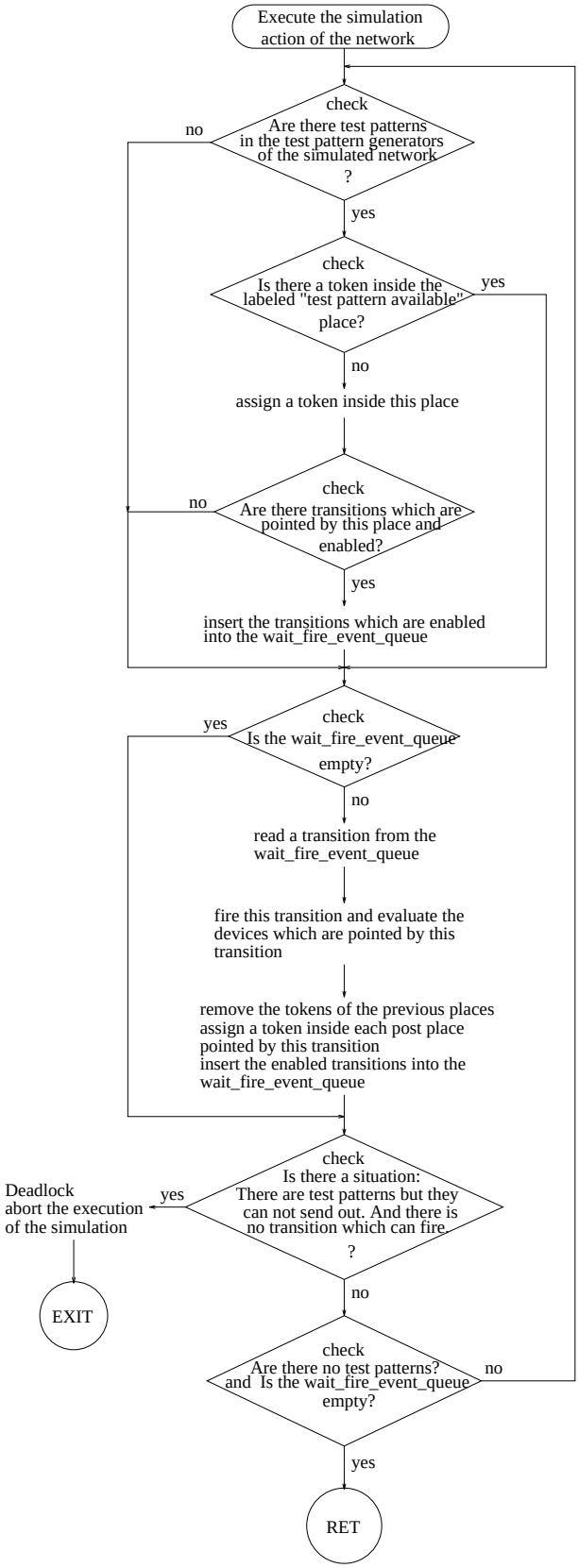


Figure 5.9: The flow chart of the simulation procedure

The mapping relation between the corresponding Petri net model and the simulated network model is shown in Figure 5.10. It shows how the Petri net model works to control the simulation execution. The tokens inside places flow through the Petri net model and therefore let the simulator simulate the behaviour of the simulated network. When the related transitions are fired, the corresponding devices will be evaluated and then produce the new output values, which will be saved into the output terminals. For example, when the transition labelled "dmuller-c:e" is fired, the dmuller-c device pointed by this transition will check both logic values of its input pins and produce the correct output value, which will be used to set the output point, i.e. the event has arrived the output of the dmuller-c module from its input.

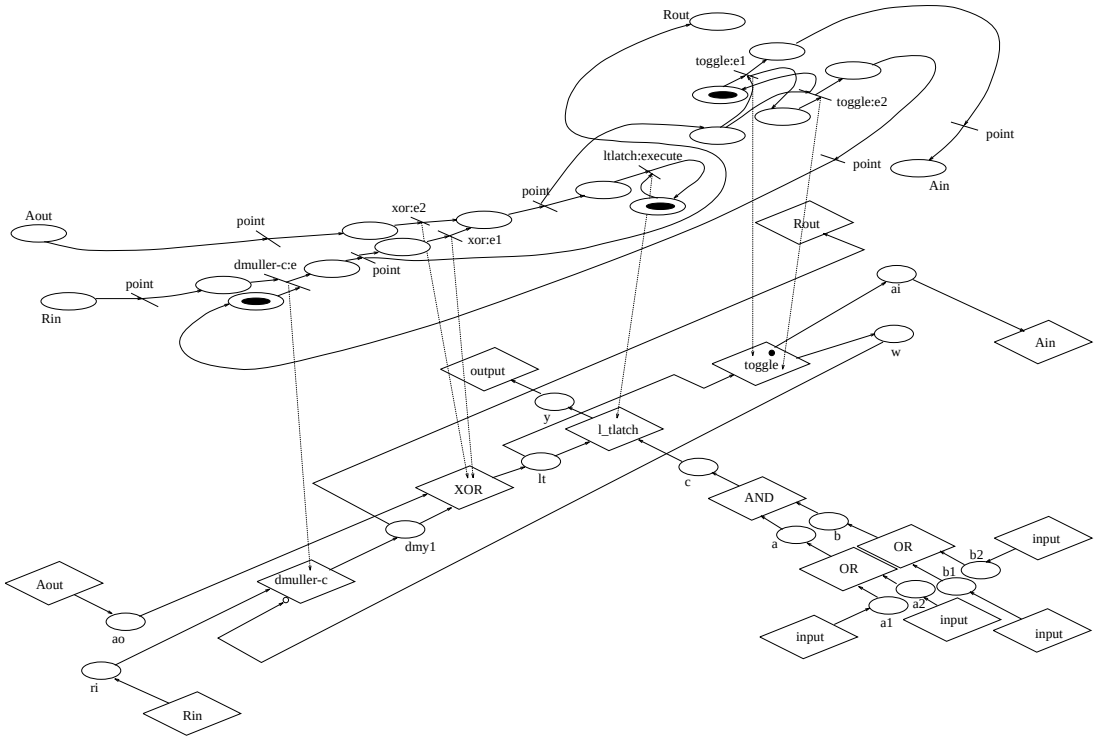


Figure 5.10: The mapping relation between the Petri net model and the simulated network model

The event functions of various Petri net transitions of the logic modules were described in Chapter 3. They will be executed when the corresponding transitions are fired.

5.6 Simulating logic devices

Within a micropipeline stage, the transparent latches are very important devices. They hold the correct logic values which are available for successor stages. Therefore, correctly modelling the behaviours of the transparent latches and letting them hold the correct values during the simulation is an important consideration. One point connected to the "lt" pin of a low-activated latch (a low-activated latch holds data if "lt" = 1) may be connected to other high-activated latches (a high-activated latch holds data if "lt" = 0) or low-activated latches simultaneously. To satisfy this situation, a state variable which denotes that the latch is transparent or holding data is necessary in each latch device model. When the "llatch1:execute" or the "hlatch1:execute" transition of the Petri net model of a low-activated or high-activated latch is fired, the latch models pointed to by this transition will check their "lt" pins. At the same time, the internal state variable denoting that the latch is transparent or holding data contains the last state. Consequently, the corresponding event function of the "llatch1:execute" or the "hlatch1:execute" transition must be executed obeying the following rule.

The evaluation rule for transparent latches:

1. If "lt" = 1, i.e. the "lt" has changed from 0 to 1, the simulator must first evaluate the logic values of the input points of all the low-activated latches connected to the same "lt" point and then evaluate the low-activated latch model letting the output points of all the low-activated latches connected

to the same "lt" point have correct output values.

2. Then the simulator changes the internal state variable denoting that the latch is transparent or holding data to its new condition.

3. Next the simulator can evaluate the logic values of the input points of all the high-activated latches connected to the same "lt" point, change the internal state variable denoting that the latch is transparent or holding data to its new condition and evaluate the high-activated latch models letting the output points of all the high-activated latches connected to the same "lt" point have correct output values.

If "lt" = 0, i.e. the "lt" has changed from 1 to 0, the positions of the low-activated latches and high-activated latches will have to be interchanged in the above evaluation rule, i.e. the simulator will first consider all the high activated latch actions and then the low-activated latch actions.

Before the simulator evaluates the logic values of the input points of a latch, it is necessary to look for all of the logic devices located on between the input point of the latch and a test pattern generator or another latch which holds data. At this search stage, all the met logic devices must be saved into the device stack and their input points must be saved into the point queue. The point queue is used to continue this search. When this search is complete, the device stack will contain all the logic devices which need to be evaluated. After these logic devices are orderly evaluated, the correct logic value will appear on the latch's input point.

5.7 Simulating delays

In addition to obtain the logic values of each point within the simulated network by correctly evaluating each device, it is also necessary to simulate the delay

time of the simulated network. For this purpose, there exists a time queue in each object of the Point class. The time queue is used to store the time values when the logic value of the point is changed. In each object of the Device class, the values of the delay time have been set. When these Device objects denoting devices are evaluated, the delay time will be added to the time values which are read from the input points of the device and these time values (sum) will be inserted into the time queues within the output points of the device. Figure 5.11 illustrates how the simulation of the delay time works. To get the correct delay time simulation, the values of the delay time of each logic device must be correctly set. These parameters can be found in the TTL or CMOS data books or technical reports of event-driven logic modules.

When the simulation work is complete, the simulator can show the waveforms by reading the time queues within each Point object.

It is necessary to check whether the data signals are slower than the control signals. To implement this part, the simulator needs to calculate the sums of the delay times of the data signal paths and the control signal paths. When the simulator evaluates a latch model, it is necessary to go back to find every device which holds fixed values. The device may be another latch, a test pattern generator or a constant. Then the last state change time of the device is read to add to all the values of the delay times of all devices which are located on the data signal paths and the control signal paths respectively. If the sum of the delay times of all the devices located on the data signal paths is bigger than that located on the control signal paths, an error message will be generated, i.e. the simulated network has some design errors. We can also use similar methods to check fan-in, fan-out, throughput ... and so on.

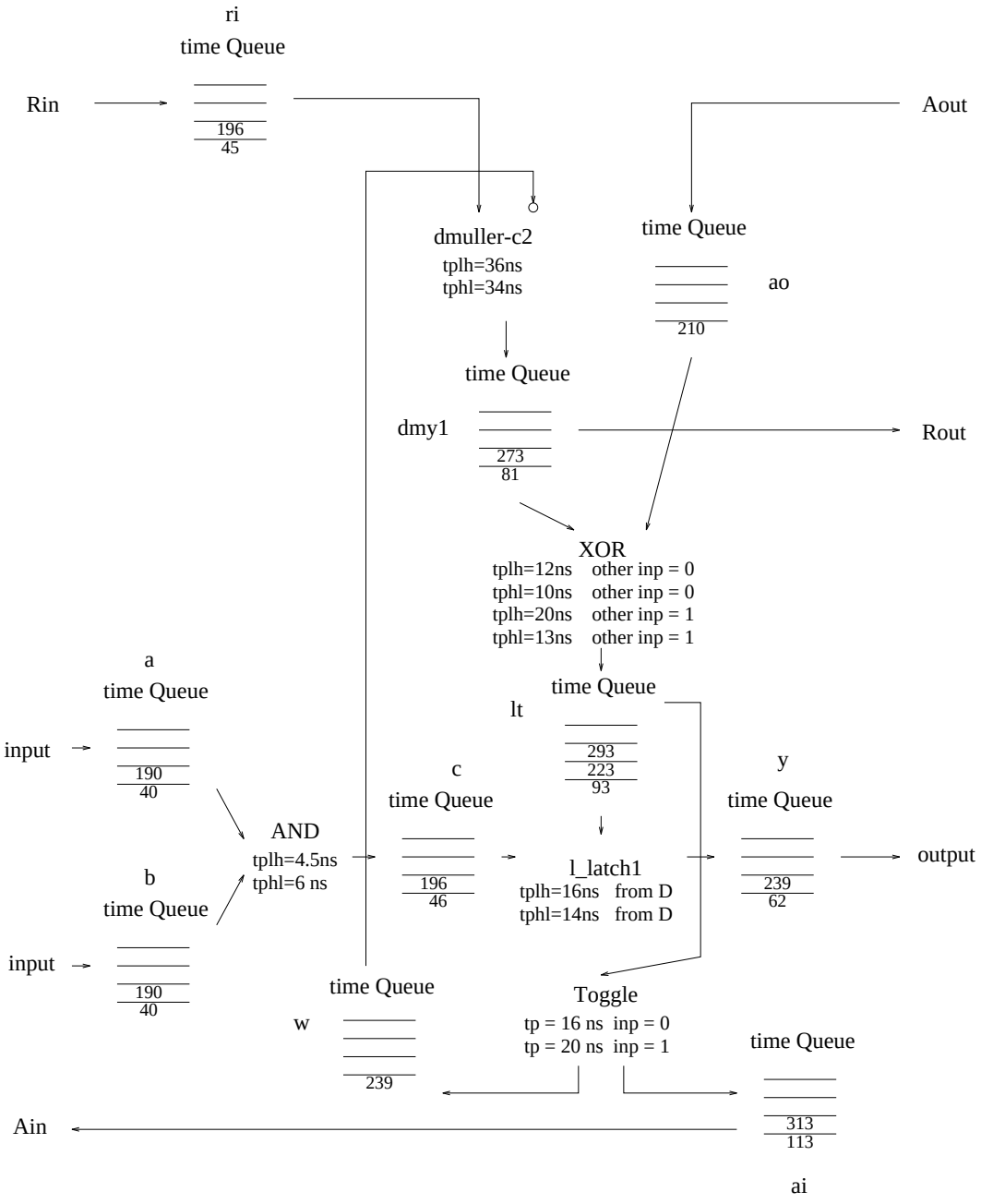


Figure 5.11: The method of recording the values of the state change time

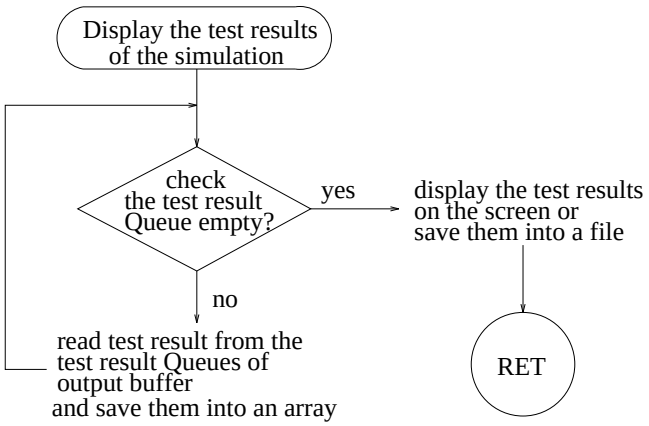


Figure 5.12: The flow chart of the function for displaying test results on the screen

5.8 Displaying the simulation results

When the simulation work is complete, the test results will be displayed on the screen and saved into an output file. Figure 5.12 shows the flow chart of the program which can be used to display the test results on the screen. The simulator only needs to read the test results from the output buffers and save them into an array and then display these test results on the screen and store them into an output file. The circuit of Figure 2.4 and the simulation descriptions and test patterns shown in Section 4.1.2 can be used to test the simulator. The simulation work is controlled by the corresponding Petri net model shown in Figure 5.7 and the simulation procedure (See Section 5.5). When all the test patterns are sent out and there is no enabled transition inside the waiting fire queue, the simulation is complete. The following test results will be obtained.

```
The results of latch#y are:

test_result: 0
test_result: 0
test_result: 0
test_result: 0
test_result: 0
test_result: 1
test_result: 1
test_result: 1
test_result: 0
test_result: 1
test_result: 1
test_result: 1
test_result: 0
test_result: 1
test_result: 1
test_result: 1

No errors. The simulation is correct.
```

If there are some mismatch errors, the error positions will be displayed. During the simulation, all the values of the state change time will be recorded in each Point object. The following values of the state change time are read from the time queues within each point object.

```
Show waves:

  the wave of latch#lt is 180
  the wave of latch#lt is 320
      .
      .
      .
  the wave of latch#lt is 4980
  the wave of latch#lt is 5120
  the wave of latch#c is 1520
  the wave of latch#c is 2480
  the wave of latch#c is 2800
  the wave of latch#c is 3760
```

```
the wave of latch#c is 4080
the wave of latch#y is 1620
the wave of latch#y is 2580
the wave of latch#y is 2900
the wave of latch#y is 3860
the wave of latch#y is 4180
the wave of latch#a1 is 2440
the wave of latch#a2 is 1160
the wave of latch#a2 is 2440
the wave of latch#a2 is 3720
the wave of latch#a is 1180
the wave of latch#b1 is 520
the wave of latch#b1 is 1160
the wave of latch#b1 is 1800
the wave of latch#b1 is 2440
the wave of latch#b1 is 3080
the wave of latch#b1 is 3720
the wave of latch#b1 is 4360
the wave of latch#b2 is 200
the wave of latch#b2 is 520
    .
    .
    .
the wave of latch#b2 is 4360
the wave of latch#b2 is 4680
the wave of latch#b is 220
the wave of latch#b is 1180
the wave of latch#b is 1500
    .
    .
    .
```

The waveforms shown in Figure 5.13 are transferred from the values of the state change time read from the time queues within each point object.

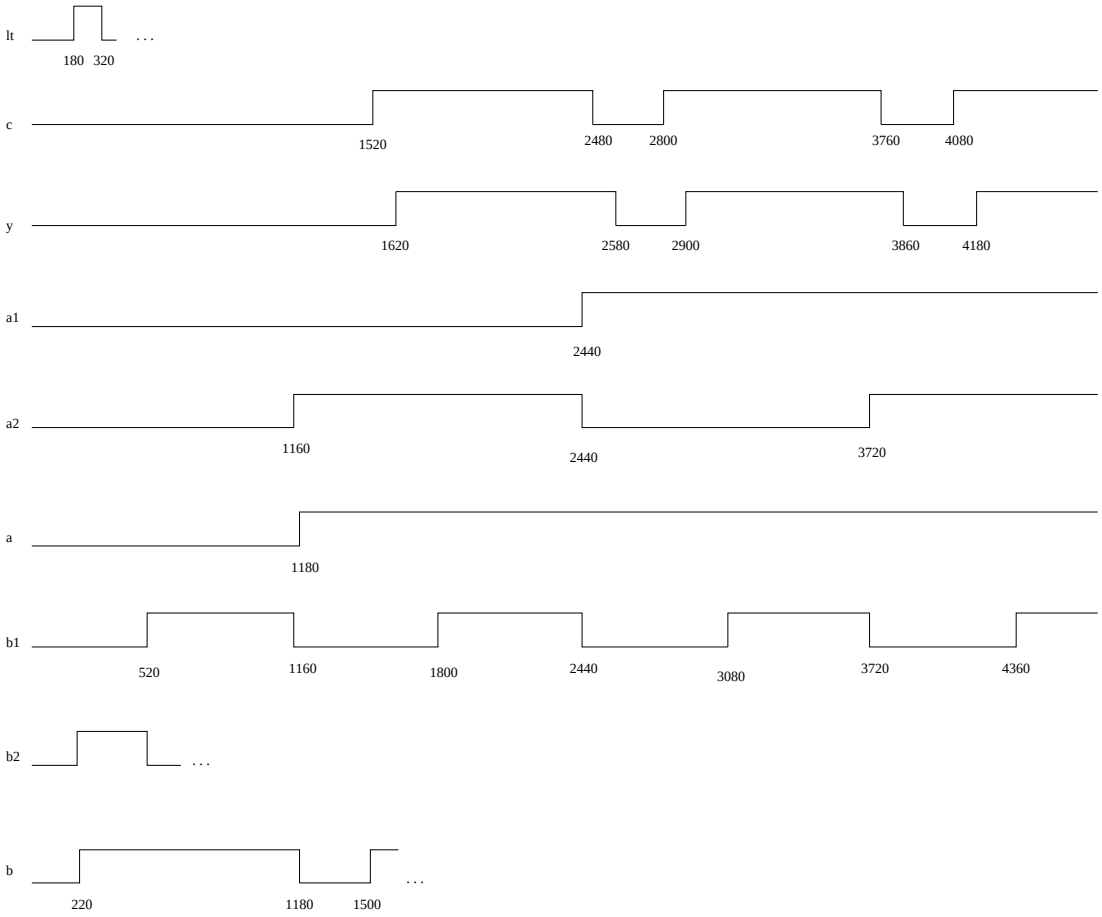


Figure 5.13: Partial waveforms of the circuit shown in Figure 2.4 after simulation

There are several other examples which are tested. They are two micropipeline stages connected in series, a micropipeline stage forking into another two micropipeline stages, a micropipeline stage forking into two micropipeline stages which join into another micropipeline stage, a two-bit multiplier micropipeline stage, a four-bit multiplier micropipeline stage, and two 2-bit multiplier micropipeline stages joining into one 4-bit multiplier micropipeline stage. All of them are presented in Appendix A.

Chapter 6

Discussion

The aim of this chapter is to analyse the performance of the micropipeline simulator which has been implemented and tested by running some examples in Chapter 5 and to discuss some problems encountered during the implementation. The simulator is able to send the test patterns flowing through those micropipelines and obtain the correct results. In addition, some design errors in the micropipelines can be discovered through the simulation easily. The first part of this chapter will examine the simulation results of the micropipelines of Figures 2.4 and A.1 using a standard hardware simulator, Silos II. Then the performance of the simulator will be discussed. From the comparison of the results and simulations, the modelling technique will be examined again and the advantages and disadvantages of the micropipeline simulator will be summarized. The comparison of the Petri net models used in this project for modelling the behaviours of the event-driven logic modules and controlling the simulation executions of micropipelines with that used in [Dill 92] for describing the specifications of the self-timed queues will also be discussed.

6.1 Simulating micropipelines using Silos II

The micropipelines in Figures 2.4 and A.1 have been simulated as described in Appendix B. From the results of the experiment, there are two issues which will be

explained in the following subsections. These are two reasons strongly in favour of the implementation of the micropipeline simulator.

6.1.1 Test pattern generators and result buffers

The waveforms of the simulation results shown in Appendix B clearly exhibit the operations of these two micropipelines. However, The "Aout" signals of the output end of these micropipelines cannot be given automatically. They must be given by the user via the test vectors. Similarly, Silos II cannot automatically check the "Ain" signals of the input end of these micropipelines to send the "Rin" signal when the next test vector is available. Therefore, when simulating micropipelines using Silos II, the user must ensure that the operation of the simulated micropipeline circuit has finished and send the "Aout" signal to the simulated circuit via the test vector. Then the user can send the next test vector and the "Rin" signal to the simulated circuit. If the "Aout" signal is sent to the simulated circuit too early, incorrect simulation results will be obtained. If the simulation environment is not ideal, a lot of time will be spent attempting various situations during the simulation. This does not mean that Silos II is not an ideal simulator. It only means that simulating micropipelines using such a simulator is not convenient. To simulate micropipelines using Silos II conveniently, a test pattern generator and a result buffer using a two-phase bundled data convention are required. If these two devices are implemented, then the simulation of micropipelines using Silos II will become easy. However, this implementation is not very simple. An interface for entering the test vectors to the test pattern generator and exporting the test results from the result buffer is also needed.

6.1.2 Event-driven logic modules

It is quite easy to design asynchronous circuits and micropipelines using event-driven logic modules. However, these event-driven logic modules are not available within the libraries of Solo 2030. Therefore, before simulating micropipelines, it is necessary to implement the schematics of these event-driven logic modules. However, the design of the event-driven logic modules is not easy. These designs must be done by professional engineers. It is not always the case that those who are interested in micropipelines are interested in and familiar with the design of event-driven logic modules. Consequently, simulating micropipelines using Silos II can be made convenient, but the library problem of event-driven logic modules must be solved.

6.2 The performance of the simulator

The micropipeline simulator makes simulating micropipelines very easy. It also provides some error detection abilities. Most importantly it can connect a test pattern generator and a result buffer to the simulated network. Therefore, test patterns can be very easily input to the simulated network from the test pattern generator and simulation results can also very easily be exported from the result buffer of the simulated network. This section will analyse the performance of the simulator.

6.2.1 An environment for simulating micropipelines

An environment for simulating micropipelines has been constructed. Designed micropipelines are able to be run conveniently within the simulator. The designed circuits do not need to be composed from actual components. The flow of the events through the circuits can be recorded during the simulation. It is very

helpful for checking the correctness of the designed micropipelines. Comparing both results of the simulation tests which are done in Chapter 5 and Appendix B individually, it is clear that the micropipeline simulator is able to do basic functional simulations of the micropipeline circuits. The test patterns can easily be read from an input file and sent into the test pattern generator which is specified to be connected to the simulated network. The test results can also easily be read from the result buffer, which is specified to be connected to the simulated network, and saved into an output file. The event-driven logic modules have been modelled using Petri nets and the C++ classes. Therefore, it is not necessary to consider the design issues of the event-driven modules. The simulation of various micropipeline circuits has been demonstrated. The simulation problems which are encountered in Silos II are not met in this micropipeline simulator.

However, the micropipeline simulator is not perfect. This is the first design stage of the simulator. There are many simulation functions which are not implemented yet. Nevertheless, the simulator's basis is Petri nets and the C++ language whose mathematical theory and object-oriented programming techniques can result in the simulator having a wide development potential.

6.2.2 Error detection

One benefit of using the simulator is that design errors within the designed micropipeline circuits can be found by this modelling technique. The design errors which may occur are errors of delay time, data computation, deadlock, infinite loop, and fan-out. They will be described in the following subsections respectively.

Delay time errors

It is possible that a micropipeline contains delay time errors, since micropipelines are not fully delay-insensitive circuits. The delay time error means that the data signal delays are longer than the request signal delays. The storage elements within each stage will catch the data appearing on their inputs when the request events arrive such storage elements. However, if the delays of the data paths are very long, then the storage elements could catch the incorrect data. Such errors affect the correctness of the designed micropipelines. Therefore, discovering delay time errors is a very important mission of the simulator. The method for discovering delay time errors has been described in Section 5.7.

Data computation errors, deadlock errors, and infinite loop errors

Three situations may happen if the connections of the control circuits of the simulated network contain errors. One is that the events can still flow through the erroneous network and there is no impact on the two-phase bundled data convention within this network. This means that such errors may be the design errors of the data computations within some stages. To discover such mistakes in micropipelines, running adequate test patterns is necessary. The second is that the events are not able to flow completely through the erroneous network. This means that some connections between stages and modules violate the two-phase bundled data convention or there are some disconnections between stages and event-driven modules within the control circuits. In this case, the deadlock will happen during the simulation. Once the deadlock happens, there is no enabled transition within the corresponding Petri net model of the simulated network. The simulator will abort the simulation work and print out the devices which are pointed to by the last firing transitions. Therefore, it is easy to find such deadlock errors in the simulation. The third is that there are some infinite loops

within stages or around some stages and new events are not able to get into the micropipeline network, i.e. infinite loop errors happen. The reason is that the terminal conditions of the control circuits are incorrectly designed. However, finding such errors is as difficult as finding infinite loop errors in a software program. This problem is not considered during the implementation because the input to the simulator is the designed circuits rather than behavioural specifications.

Fan-out errors

It is impossible for a device to produce an infinite output current. Thus, there is a limitation of each device (i.e. the "fan-out"), which represents how many output devices a device is able to connect to. For example, in the two-bit multiplier micropipeline stage of Figure A.8, there are three 2-input XOR circuits. One connects all the transparent latches (including high activated latches and low activated latches) and one connects all the 2-to-1 multiplexers. It is necessary to check whether such two XOR circuits have sufficient output currents which are able to support all the transparent latches or all the 2-to-1 multiplexers and cause the incorrect output voltages of such two XOR circuits.

All these parameters can be set in the Device objects of the circuit model which is constructed by the micropipeline simulator. Therefore, the "fan-out" will be checked when the events or data values arrive each Point object during the simulation. If the simulated network contains such errors, they will be printed out when the simulation is complete. Hence, such errors can be found easily using the micropipeline simulator.

6.2.3 Performance measurements

There are two important parameters for a pipeline, which are the throughput and the latency. These two parameters can be measured using this modelling technique. The methods of measuring them are described in the following subsections.

Throughput

Figure 5.11 has shown a method for recording the values of the state change time. Whichever micropipeline stage within the simulated network is needed to measure its throughput, the following procedure can be applied. First, the occurrence time of the earliest "Rin" event which is recorded in the time queue of the Point object for denoting the input terminal of the "Rin" signals must be found. Suppose that it is denoted by t_1. Second, the occurrence time of the last "Rin" event must be found. Suppose that it is denoted by t_2. Third, the occurrence number of "Rin" events which are recorded in the time queue of the Point object for denoting the input terminal of the "Rin" signals must be found. Suppose that it is denoted by n. Thus, the throughput can be obtained by

$$n \ / \ (\ t_2 - t_1 \ ),$$

i.e. the throughput is equal to the frequency of the "Rin" signals. The throughput can also be obtained from measuring the frequency of the "Rout" signals. Measuring the throughput of the different stages only needs changing the Point objects of the above description, where the time queues contain the t_1, t_2 and the n can be found. Not only the throughput of each stage but also the entire simulated network can be measured using the same method.

Latency

A similar method can be used to measure the latency of a micropipeline stage within the simulated network. First, the occurrence time of the earliest "Rin" event which is recorded in the time queue of the Point object for denoting the input terminal of the "Rin" signals must be found. Suppose that it is denoted by $t_1(1)$. Second, the occurrence time of the corresponding "Rout" event which is recorded in the time queue of the Point object for denoting the output terminal of the "Rout" signals must be found. Suppose that it is denoted by $t_2(1)$. Third, the subsequent time pairs of "Rin" and corresponding "Rout" must continually be found. They are denoted by $t_1(2)$, $t_2(2)$, $t_1(3)$, $t_2(3)$, ... and so on. Fourth, the occurrence number of "Rin" events which are recorded in the time queue of the Point object for denoting the input terminal of the "Rin" signals must be found. Suppose that it is denoted by n. Then, the latency can be obtained by

$$(\sum_{i=1}^{n} (t_2(i) - t_1(i))) / n,$$

i.e. the latency is equal to the average of the occurrence time difference of the corresponding "Rout" event and the "Rin" event. Measuring the latency of the different stages only needs changing the Point objects of the above description, where the time queues contain the t_1, t_2 and the n can be found. Not only the latency of each stage but also the entire simulated network can be measured using the same method.

6.2.4 Future developments

Many debugging functions and the graphical I/O interface are not implemented yet because this simulator is an experimental prototype. However, a user friendly interface for entering the simulated network and powerful libraries which should contain the standard logic elements and various event-driven logic modules will be considered in next design stage. The internal data representations are the

objects of the C++ classes. Any required data which need to be observed during the simulation can easily be added into the related objects in the future. When adding new information to the related C++ classes, the problems of memory space will be considered. Unnecessary information will be removed. Pointers are used to link an object to others in the circuit models and Petri net models within the simulator. However, object-oriented techniques will be considered to reduce the linkages between the objects. These interesting issues are pointed out here. The author hopes they can be followed by other interested researchers.

6.2.5 Advantages and disadvantages

This section will summarize the advantages and disadvantages of the simulator and the modelling technique. The advantages are illustrated as follows:

- The simulator provides an environment for simulating micropipelines. Thus, the test patterns can easily be entered into the simulated network and the test results can also be displayed on the screen or saved into an output file.

- The values of the state change times can be recorded in the time queues within each Point object during the simulation. Thus, the errors and the performance of the simulated network can be obtained from analysing such time values.

- The event-driven logic modules have been modelled by Petri nets and C++ classes, denoted by some abbreviations. Users do not need to encounter the design problems of the event-driven modules and can immediately begin simulation work on their micropipeline designs.

- This modelling technique and the simulator are useful for learning about micropipelines. The corresponding Petri net model of each micropipeline circuit can be constructed. Therefore, analysing the micropipeline becomes

very easy. The flow of tokens through the Petri net model is like the flow of events through the actual micropipeline circuit. It is very helpful for those who want to understand the operation of micropipelines. Therefore, the simulator is also a good demonstration tool for micropipelines.

The disadvantages are also illustrated as follows:

- The simulator has no user friendly interface for entering the simulated network and displaying the simulation results.
- The simulator has no library of standard logic elements.
- The problems of memory space are not considered in this modelling technique.
- The design errors of micropipelines are not considered in this modelling technique.

6.3 Some implementation problems

There are three implementation problems worth explaining in this section. They are modelling transparent latches, modelling data signal flow, and comparing with Dill's Petri nets [Dill 92]. The author has spent a lot time debugging the first problem and solving it during the implementation. Therefore, it is particularly emphasized here to remind other researchers. There are two methods for modelling the data signal flow. One is the **forward-set-up** method. The other is the **backward-look-for** method. The **backward-look-for** method is used in the project. Petri nets are used to describe the specification of the self-timed queues in [Dill 92]. The use of Petri nets there is different from that used in this project. These issues will be discussed in the next subsections individually.

6.3.1 Modelling transparent latches

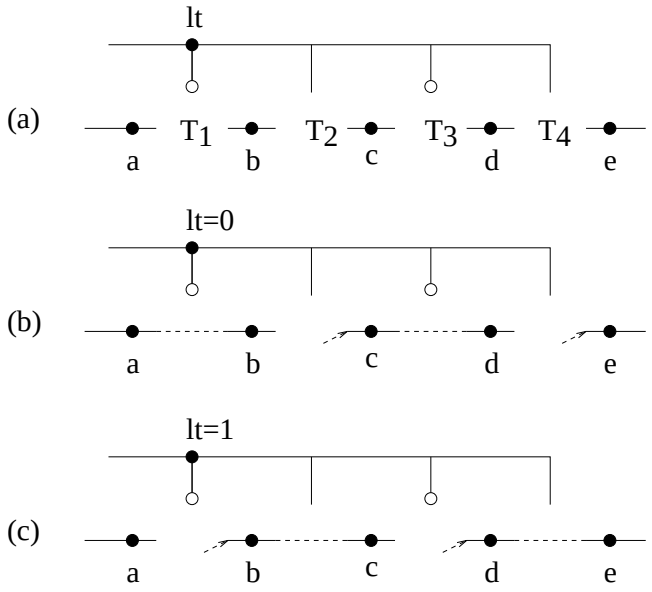


Figure 6.1: Modelling transparent latches

Figure 6.1(a) shows four transparent latches connected in series with their "lt" inputs connected to the point labelled "lt". Figures 6.1(b) and 6.1(c) show the states of the latches when **lt** is equal to **0** (logic "low") and **1** (logic "high") respectively. The rectangle boxes within the transparent latches denote the internal states of the latches, i.e., the previous logic values. If **lt** is equal to **1**, i.e., **lt** has gone from **0** to **1**, the states of the four latches are changed from their states in Figure 6.1(b) to the states in Figure 6.1(c). Latch T_1 holds the original data of point **a**, i.e., **b = original a**. Latch T_3 holds the original data of point **c**, i.e., **d = original c**. Point **c** will be equal to **b**, i.e., the original value on point **a** will appear on point **c**. Similarly, the original value on point **c** will appear on point **e**. The data have propagated through these four latches. When the value of **lt** becomes **0**, i.e., **lt** has gone from **1** to **0**, the states of the four latches are changed from the state shown in Figure 6.1(c) to the state of Figure 6.1(b). Latch T_2 holds the original data of point **b**, i.e., **c = original b**. Latch T_4 holds the

original data of point **d**, i.e., **e = original d**. Point **d** will be equal to **c**, i.e., the most original value on **a** will appear on point **d**. To correctly model such latches, it is necessary to obey **the evaluation rule for transparent latches** which has been described in Section 5.6. This evaluation rule for transparent latches has been tested by the examples of A.4, A.5, and A.6. When these examples are simulated, the correct results are obtained.

6.3.2 Modelling data signal flow

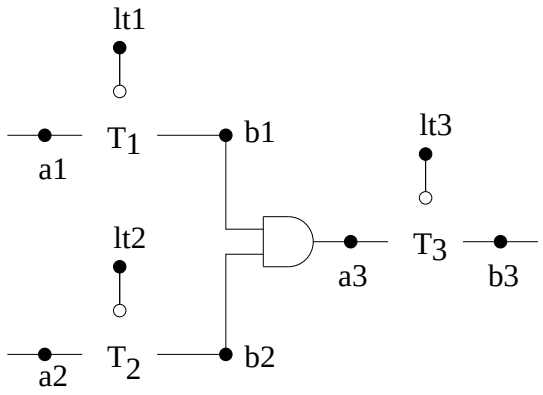


Figure 6.2: Modelling data signal flow

Figure 6.2 shows three transparent latches. Their "lt" inputs are connected to point "lt1", point "lt2", and point "lt3" respectively. The **forward-set-up** method is that if any element controlled by events, such as multiplexers, transparent latches, and test pattern generators, changes its output values, these new output values must be sent to each reachable point within the simulated network. The reachable points include all the points whose positions are located between such element and isolated elements. The isolated elements are transparent latches which hold previous data values and multiplexers whose outputs are connected to other inputs, i.e., their inputs (reachable points) are not connected to their outputs (unreachable points). The logic devices between such element of changing

its output values and those isolated elements will be orderly evaluated. Then each reachable point will contain the correct value. For example, if latch T_1 changes its output value (i.e. the value of point **b1**) and latch T_3 holds the fixed value, point **a3** is a reachable point and latch T_3 is an isolated element. Thus, the AND gate will be evaluated and then point **a3** will contain the correct value.

The **backward-look-for** method is that if any element controlled by events receives an event, it is necessary to go back from the element's inputs to look for each element holding fixed values, such as latches, test pattern generators, and constant elements. All logic devices between the element and those elements holding fixed values must be evaluated. Then the element is also evaluated and the correct output values will be set into its output points. For example, if latch T_3 receives an event and latches T_1 and T_2 hold fixed values, going back to look for each element holding fixed values from point **a3** is made. Thus, latches T_1 and T_2 are found. Next, the AND gate will be evaluated to get the correct value of point **a3**. Then latch T_3 will be evaluated to obtain the correct output value of point **b3**.

However, whether either of the above two methods is used, **the evaluation rule for transparent latches** must be obeyed when the latch is evaluated.

6.3.3 Comparing with Dill's Petri nets

Dill's Petri net models are used to describe the behavioural specifications of the self-timed queue whose operations are based on a two-phase handshaking protocol [Dill 92]. Sufficient tokens inside the places within Dill's models are used to represent that the events (request or acknowledge) may arise. The tokens flowing through his Petri net models are like the conditions changing, and firing a transition gives rise to events propagating through the behaviour models rather than the actual circuits. In this modelling technique, the tokens inside the places

within the Petri net models constructed by the micropipeline simulator are used to represent events (request or acknowledge). The tokens flowing through the Petri net models are like the events flowing through the control circuits of the micropipelines. Dill's use of Petri nets is similar to the "dual" [Peterson 81] of the Petri net model used here. However, both Petri net models make different contributions to modelling two-phase handshaking asynchronous circuits. In this research, the data signal delay problem has also been considered. The Petri net model is used to model fully event-driven logic modules and to control the simulation execution of the micropipeline simulator.

Chapter 7

Conclusions and further work

7.1 Conclusions

From this experiment, it is believed that asynchronous circuits that use this two-phase handshake protocol will have the property of high performance and low power. Micropipelines consist of event-driven modules and event-controlled storage elements that are designed for two-phase transition signalling. The design time and cost will be reduced when micropipeline techniques and event-driven modules are used to design complex asynchronous systems. These techniques and modules are being used to develop microprocessing systems of high performance and low power.

A basic micropipeline simulator based on Petri net models has been implemented in this research work. The implementation of the simulator clearly proves that Petri net models are useful for studying the behaviour of micropipelines and discovering the design problems. It can also be emphasized that the C++ classes are useful for modelling asynchronous systems. Another benefit of using C++ language is that it is not possible to know how many devices and terminals a simulated network has in advance. However, the free storage of the C++ provides the ability of the dynamic memory allocation. It makes the implementation of variable size lists for any input data, such as devices, stages, places, transitions ...

etc., very easy. Therefore, the contributions of this research work are illustrated as follows:

- It provides a better environment for simulating micropipelines.
- The modelling technique is useful for learning about micropipelines. Particularly, the corresponding Petri net models of micropipelines clearly demonstrate the behaviour of events through the actual micropipeline circuits.
- The modelling technique can be used to discover some mistakes of micropipelines, such as whether data signal delays are longer than control signal delays.
- The modelling technique can also be used to measure the performance of micropipelines, such as throughput and latency.

7.2 Further work

Modelling techniques are necessary for studying micropipeline asynchronous circuits in order to evaluate high-level specifications, to synthesize low level circuits, and to simulate the circuits after synthesis. To achieve a perfect micropipeline simulator for designing such circuits, the author would continue to improve the simulator's functions and make it able to do simulations and analyses of more complex micropipeline systems. For this purpose, establishing a library of logic devices and designing a user-friendly interface are required.

Although Petri net theory [Peterson 81] is available, it is still necessary to develop the mathematical theory for the corresponding Petri net models of micropipeline designs. Then design errors of micropipelines can be found through analysing their corresponding Petri net models.

Within micropipeline circuits data signal delays should be shorter than request signal delays. Special delays are sometimes required in the control path when significant processing logic is put between storage elements in the data path. However, careful partitioning of the logic functions of the system into several stages can improve on the system to have the optimum performance. The performance of micropipeline designs can be evaluated using this modelling technique. Therefore, it is worth studying how the micropipeline designs can be converted into their optimum counterparts using this modelling technique.

Appendix A

Some test examples

Chapter 5 describes the implementation of a micropipeline simulator. In this Appendix there are several examples which will be used to test this simulator. They include two micropipeline stages connected in series, a micropipeline stage forking into two micropipeline stages, a micropipeline stage forking into two micropipeline stages which join one single micropipeline stage, a two-bit multiplier micropipeline stage, a four-bit multiplier micropipeline stage, and two 2-bit multiplier micropipeline stages joining into one 4-bit multiplier micropipeline stage. These examples and their simulations will be described in the following sections individually.

A.1 Two serially connected stages

The network shown in Figure A.1 consists of two stages and can be described as follows.

```
stage: stg1,
  llatch1: lt, c, y,
  or2: a1, a2, a,
  or2: b1, b2, b,
  and2: a, b, c,
  dmuller-c2: ri, w, dmy1,
  mxor2: dmy1, ao, lt,
  toggle: lt, ai, w,
  rin: ri,
  ain: ai,
  rout: dmy1,
```

```
    aout: ao,
    input: a1,
    input: a2,
    input: b1,
    input: b2,
    output: y,
  stage: stg2,
    llatch1: llt, cc, yy,
    not: aa, cc,
    dmuller-c2: rri, ww, ddmy1,
    mxor2: ddmy1, aao, llt,
    toggle: llt, aai, ww,
    rin: rri,
    ain: aai,
    rout: ddmy1,
    aout: aao,
    input: aa,
    output: yy,
  network: project,
    line: stg1#y, stg2#aa,
    line: stg1#dmy1, stg2#rri,
    line: stg2#aai, stg1#ao,
```

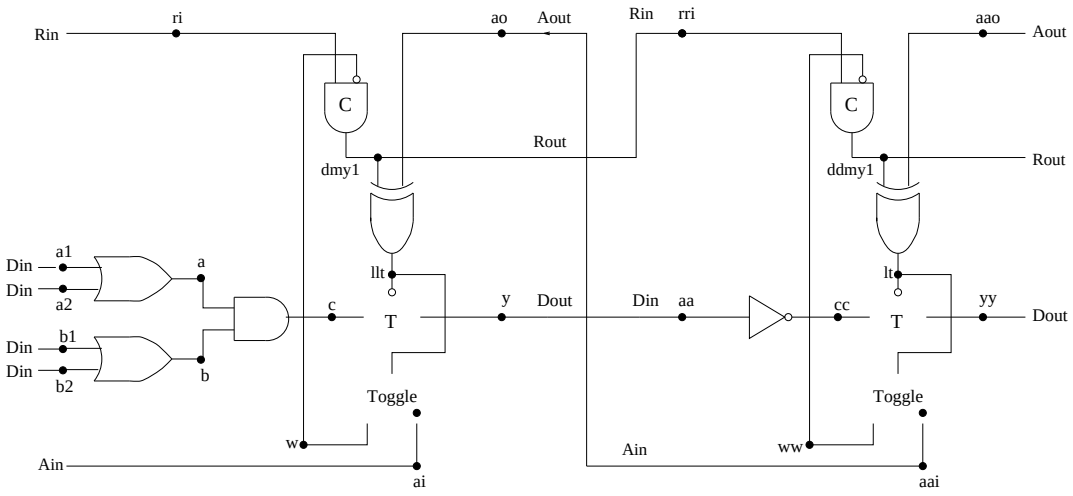


Figure A.1: An example of two micropipeline stages connected in series

The following simulation description can be used to enter the test patterns and simulate the circuit of Figure A.1.

```
definput: stg1#a1, stg1#a2, stg1#b1, stg1#b2,
defrin: stg1#ri,
defain: stg1#ai,
defoutput: stg2#yy,
defrout: stg2#ddmy1,
defaout: stg2#aao,
defformat: stg1#a1, stg1#a2, stg1#b1, stg1#b2, stg2#yy,
deftest:
xv: 0 0 0 0      1
xv: 0 0 0 1      1
xv: 0 0 1 0      1
xv: 0 0 1 1      1
xv: 0 1 0 0      1
xv: 0 1 0 1      0
xv: 0 1 1 0      0
xv: 0 1 1 1      0
xv: 1 0 0 0      1
xv: 1 0 0 1      0
xv: 1 0 1 0      0
xv: 1 0 1 1      0
xv: 1 1 0 0      1
xv: 1 1 0 1      0
xv: 1 1 1 0      0
xv: 1 1 1 1      0
endtest:
```

The corresponding Petri net model of the network of Figure A.1 is shown in Figure A.2.

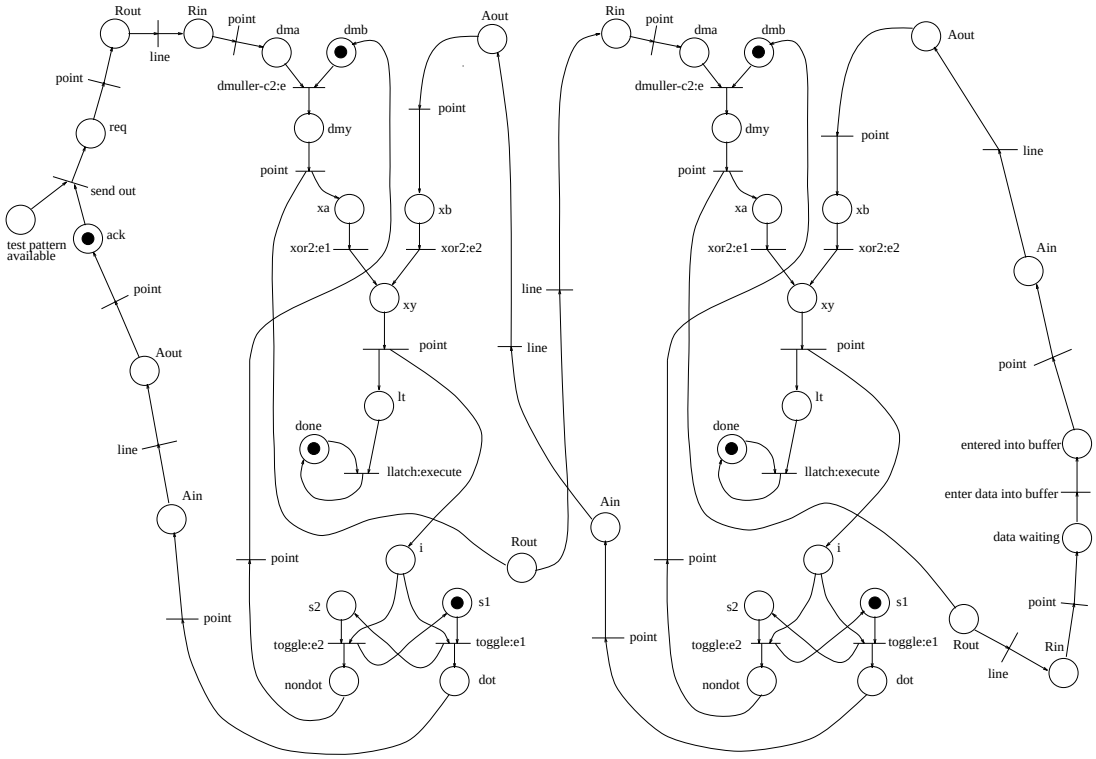


Figure A.2: The corresponding Petri net model of the above example of two micropipeline stages connected in series

The results of the simulation are as follows:

```
The test results:

        yy

         1
         1
         1
         1
         1
         0
         0
         0
         1
         0
         0
         0
         1
         0
         0
         0
```

A.2 A micropipeline forking example

The network shown in Figure A.3 consists of three micropipeline stages. They are connected to be that one stage is forked into another two stages. This network can also be described by the following descriptions:

```
stage: stg1,
  llatch1: lt, c, y,
  or2: a1, a2, a,
  or2: b1, b2, b,
  and2: a, b, c,
  dmuller-c2: ri, w, dmy1,
  mxor2: dmy1, ao, lt,
  toggle: lt, ai, w,
  rin: ri,
```

```
    ain: ai,
    rout: dmy1,
    aout: ao,
    input: a1,
    input: a2,
    input: b1,
    input: b2,
    output: y,
  stage: stg2,
    llatch1: llt, cc, yy,
    not: aa, cc,
    dmuller-c2: rri, ww, ddmy1,
    mxor2: ddmy1, aao, llt,
    toggle: llt, aai, ww,
    rin: rri,
    ain: aai,
    rout: ddmy1,
    aout: aao,
    input: aa,
    output: yy,
  stage: stg3,
    llatch1: llt3, cc3, yy3,
    not: aa3, cc3,
    dmuller-c2: rri3, ww3, ddmy13,
    mxor2: ddmy13, aao3, llt3,
    toggle: llt3, aai3, ww3,
    rin: rri3,
    ain: aai3,
    rout: ddmy13,
    aout: aao3,
    input: aa3,
    output: yy3,
  network: project,
    line: stg1#y, stg2#aa,
    line: stg1#y, stg3#aa3,
    line: stg1#dmy1, stg2#rri,
    line: stg1#dmy1, stg3#rri3,
    muller-c2: stg2#aai, stg3#aai3, stg1#ao,
```

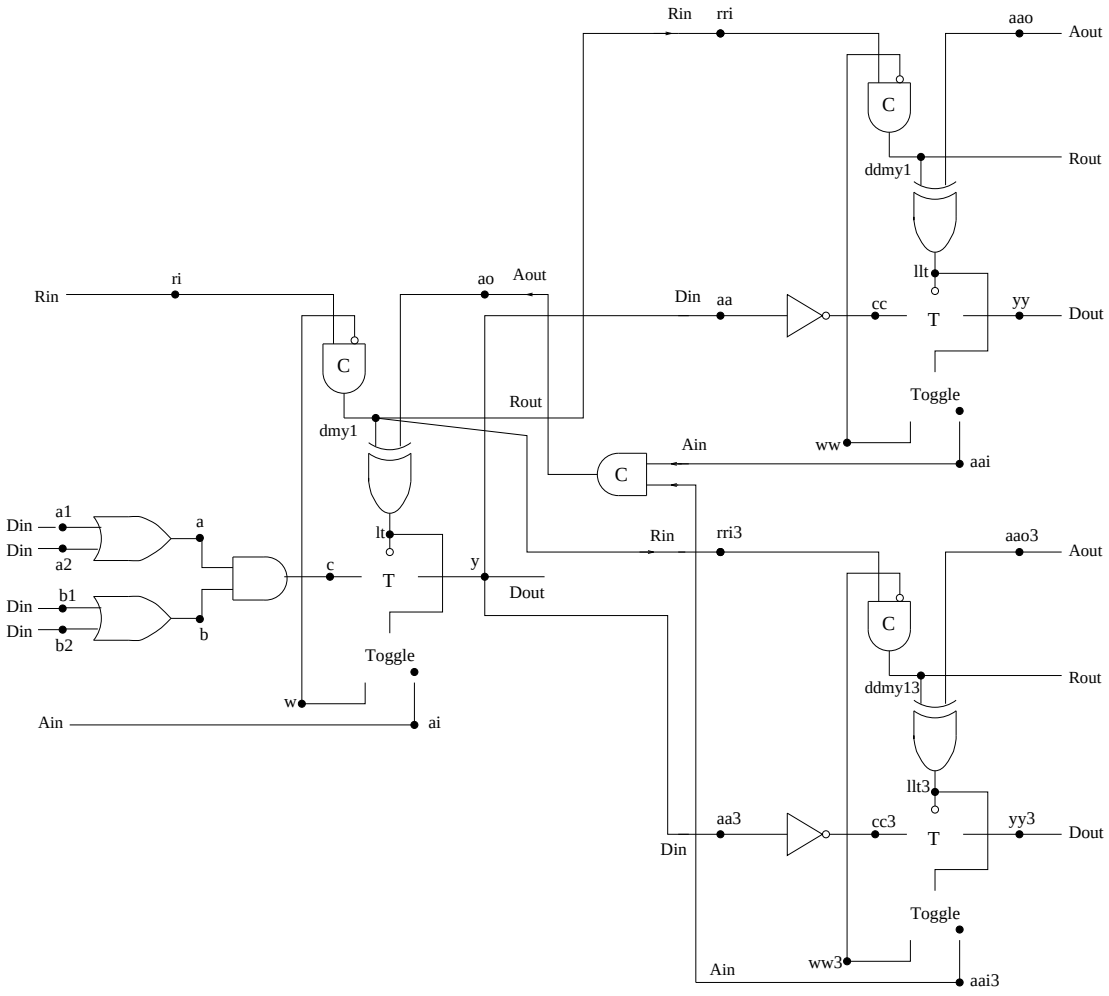


Figure A.3: A micropipeline stage forks into two micropipeline stages

The following simulation descriptions can be used to enter the test patterns and simulate the circuit of Figure A.3.

```
definput: stg1#a1, stg1#a2, stg1#b1, stg1#b2,
defrin: stg1#ri,
defain: stg1#ai,
defoutput: stg2#yy,
defrout: stg2#ddmy1,
defaout: stg2#aao,
defoutput: stg3#yy3,
defrout: stg3#ddmy13,
defaout: stg3#aao3,
defformat: stg1#a1, stg1#a2, stg1#b1, stg1#b2,
           stg2#yy, stg3#yy3,
deftest:
xv: 0 0 0 0      1 1
xv: 0 0 0 1      1 1
xv: 0 0 1 0      1 1
xv: 0 0 1 1      1 1
xv: 0 1 0 0      1 1
xv: 0 1 0 1      0 0
xv: 0 1 1 0      0 0
xv: 0 1 1 1      0 0
xv: 1 0 0 0      1 1
xv: 1 0 0 1      0 0
xv: 1 0 1 0      0 0
xv: 1 0 1 1      0 0
xv: 1 1 0 0      1 1
xv: 1 1 0 1      0 0
xv: 1 1 1 0      0 0
xv: 1 1 1 1      0 0
endtest:
```

The corresponding Petri net model of the network of Figure A.3 is shown in Figure A.4.

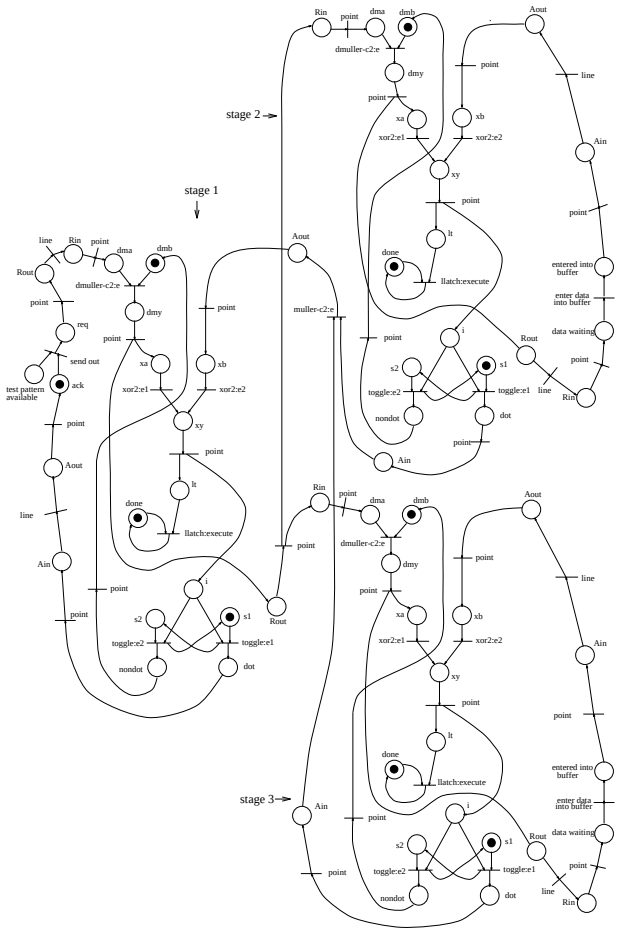


Figure A.4: The corresponding Petri net model of one micropipeline stage forking into two micropipeline stages

The results of the simulation are as follows:

```
The test results:

      yy yy3

       1  1
       1  1
       1  1
       1  1
       1  1
       0  0
       0  0
       0  0
       1  0
       0  0
       0  0
       0  0
       1  0
       0  0
       0  0
       0  0
```

A.3 A micropipeline forking and joining example

The network shown in Figure A.5 consists of four micropipeline stages. They are connected in such a way that one stage is forked into two other stages and then these two stages are joined into the fourth stage. The following descriptions can be used to describe this network.

```
stage: stg1,
  llatch1: lt, a, y1,
  llatch1: lt, b, y2,
  nor2: a1, a2, a,
  nor2: b1, b2, b,
```

```
  dmuller-c2: ri, w, dmy1,
  mxor2: dmy1, ao, lt,
  toggle: lt, ai, w,
  rin: ri,
  ain: ai,
  rout: dmy1,
  aout: ao,
  input: a1,
  input: a2,
  input: b1,
  input: b2,
  output: y1,
  output: y2,
stage: stg2,
  llatch1: llt, cc, yy,
  not: aa, cc,
  dmuller-c2: rri, ww, ddmy1,
  mxor2: ddmy1, aao, llt,
  toggle: llt, aai, ww,
  rin: rri,
  ain: aai,
  rout: ddmy1,
  aout: aao,
  input: aa,
  output: yy,
stage: stg3,
  llatch1: llt3, cc3, yy3,
  not: aa3, cc3,
  dmuller-c2: rri3, ww3, ddmy13,
  mxor2: ddmy13, aao3, llt3,
  toggle: llt3, aai3, ww3,
  rin: rri3,
  ain: aai3,
  rout: ddmy13,
  aout: aao3,
  input: aa3,
  output: yy3,
stage: stg4,
  llatch1: llt4, cc4, yy4,
  and2: c1, c2, cc4,
  dmuller-c2: rri4, ww4, ddmy14,
  mxor2: ddmy14, aao4, llt4,
  toggle: llt4, aai4, ww4,
  rin: rri4,
```

```
    ain: aai4,
    rout: ddmy14,
    aout: aao4,
    input: c1,
    input: c2,
    output: yy4,
  network: project,
    line: stg1#y1, stg2#aa,
    line: stg1#y2, stg3#aa3,
    line: stg1#dmy1, stg2#rri,
    line: stg1#dmy1, stg3#rri3,
    muller-c2: stg2#aai, stg3#aai3, stg1#ao,
    line: stg2#yy, stg4#c1,
    line: stg3#yy3, stg4#c2,
    muller-c2: stg2#ddmy1, stg3#ddmy13, stg4#rri4,
    line: stg4#aai4, stg2#aao,
    line: stg4#aai4, stg3#aao3,
```

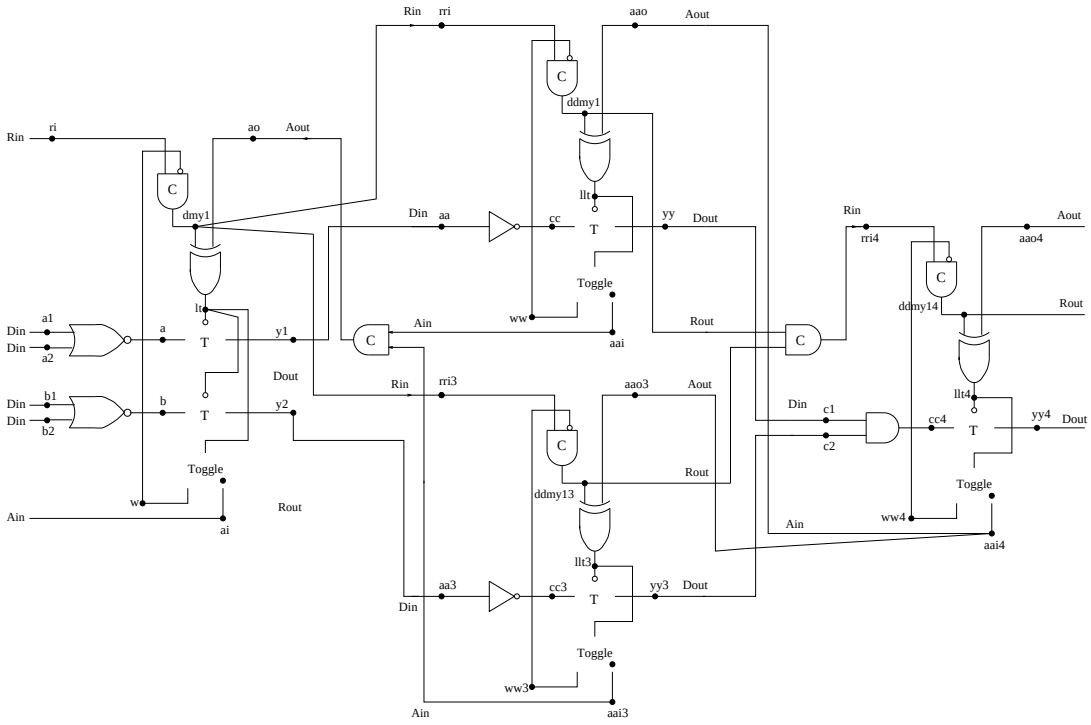


Figure A.5: A micropipeline stage forks into two micropipeline stages which join into a single micropipeline stage

The following descriptions can be used to simulate the circuit of Figure A.5.

```
definput: stg1#a1, stg1#a2, stg1#b1, stg1#b2,
defrin: stg1#ri,
defain: stg1#ai,
defoutput: stg4#yy4,
defrout: stg4#ddmy14,
defaout: stg4#aao4,
defformat: stg1#a1, stg1#a2, stg1#b1, stg1#b2, stg4#yy4,
deftest:
xv: 0 0 0 0      0
xv: 0 0 0 1      0
xv: 0 0 1 0      0
xv: 0 0 1 1      0
xv: 0 1 0 0      0
xv: 0 1 0 1      1
xv: 0 1 1 0      1
xv: 0 1 1 1      1
xv: 1 0 0 0      0
xv: 1 0 0 1      1
xv: 1 0 1 0      1
xv: 1 0 1 1      1
xv: 1 1 0 0      0
xv: 1 1 0 1      1
xv: 1 1 1 0      1
xv: 1 1 1 1      1
endtest:
```

The corresponding Petri net model of the network of Figure A.5 is shown in Figure A.6. When the simulation is complete, the correct results are obtained as expected.

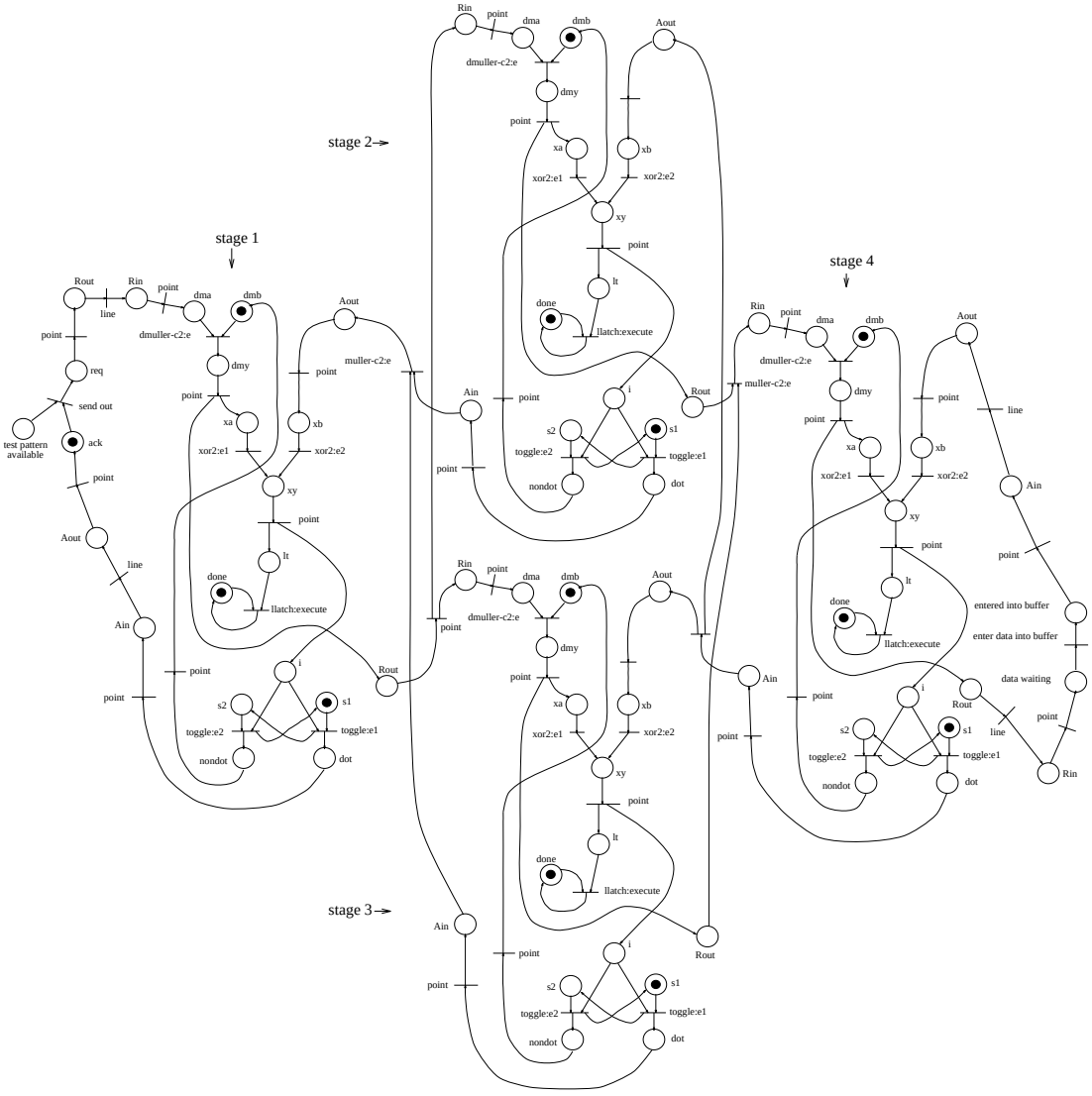


Figure A.6: The corresponding Petri net model of the above example

A.4 Two-bit multiplier example

The two bit multiplier micropipeline stage [Day 91] is shown in Figure A.7. It is more complex than previous examples. The corresponding Petri net model of this circuit is shown in Figure A.8. The simulation descriptions are shown as follows:

```
definput: multiplier#B1, multiplier#B0,
          multiplier#A1, multiplier#A0,
defoutput: multiplier#sum3, multiplier#sum2,
           multiplier#sum1, multiplier#sum0,
defrin: multiplier#dma1,
defain: multiplier#xa1,
defrout: multiplier#xb1,
defaout: multiplier#dmb1,
defformat: multiplier#B1, multiplier#B0,
           multiplier#A1, multiplier#A0,
           multiplier#sum3, multiplier#sum2,
           multiplier#sum1, multiplier#sum0,
deftest:
xv: 0 0 0 0      0 0 0 0
xv: 0 0 0 1      0 0 0 0
xv: 0 0 1 0      0 0 0 0
xv: 0 0 1 1      0 0 0 0
xv: 0 1 0 0      0 0 0 0
xv: 0 1 0 1      0 0 0 1
xv: 0 1 1 0      0 0 1 0
xv: 0 1 1 1      0 0 1 1
xv: 1 0 0 0      0 0 0 0
xv: 1 0 0 1      0 0 1 0
xv: 1 0 1 0      0 1 0 0
xv: 1 0 1 1      0 1 1 0
xv: 1 1 0 0      0 0 0 0
xv: 1 1 0 1      0 0 1 1
xv: 1 1 1 0      0 1 1 0
xv: 1 1 1 1      1 0 0 1
endtest:
```

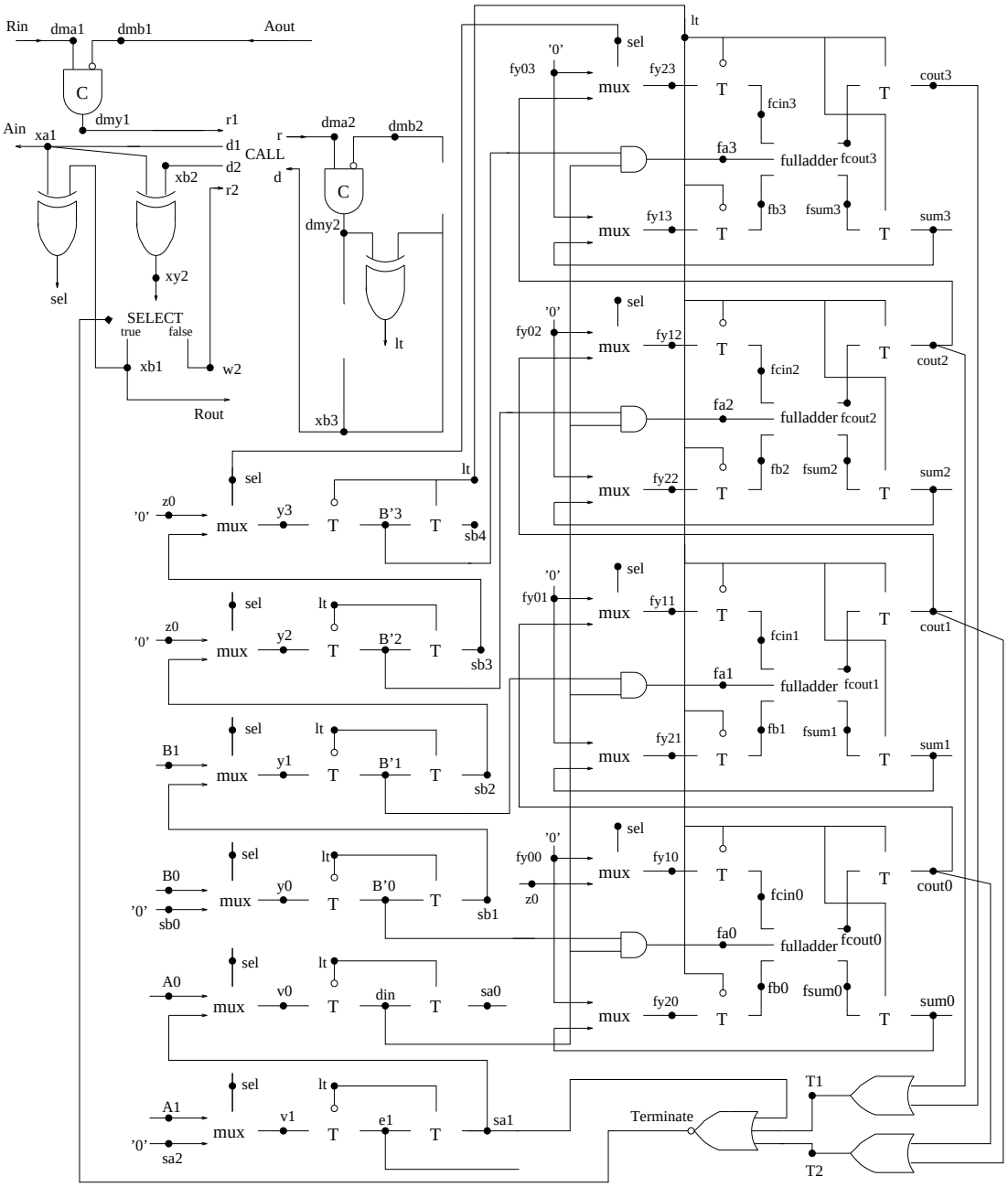


Figure A.7: A two-bit multiplier micropipeline stage

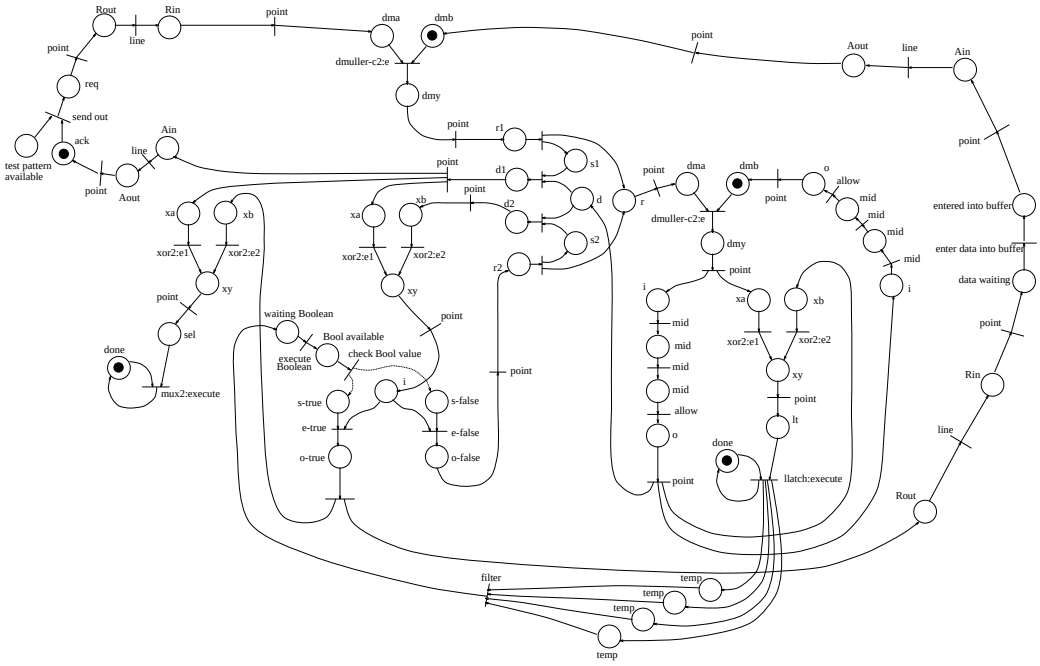


Figure A.8: The corresponding Petri net model of the above multiplier stage

The test results are shown as follows:

```
The test results:

                sum3 sum2 sum1 sum0

( 0 x 0 )    0    0    0    0
( 0 x 1 )    0    0    0    0
( 0 x 2 )    0    0    0    0
( 0 x 3 )    0    0    0    0
( 1 x 0 )    0    0    0    0
( 1 x 1 )    0    0    0    1
( 1 x 2 )    0    0    1    0
( 1 x 3 )    0    0    1    1
( 2 x 0 )    0    0    0    0
( 2 x 1 )    0    0    1    0
( 2 x 2 )    0    1    0    0
( 2 x 3 )    0    1    1    0
( 3 x 0 )    0    1    1    0
( 3 x 1 )    0    0    1    1
( 3 x 2 )    0    1    1    0
( 3 x 3 )    1    0    0    1
```

A.5 Four-bit multiplier example

The four bit multiplier micropipeline stage [Day 91] is shown in Figure A.9. The following test patterns can be used to test the simulator using almost the same Petri net model as Figure A.9 to simulate its behaviour. When the simulation is complete, the correct results are obtained as expected.

```
definput: multiplier#B3, multiplier#B2,
          multiplier#B1, multiplier#B0,
          multiplier#A3, multiplier#A2,
          multiplier#A1, multiplier#A0,
defoutput: multiplier#sum7, multiplier#sum6,
           multiplier#sum5, multiplier#sum4,
           multiplier#sum3, multiplier#sum2,
           multiplier#sum1, multiplier#sum0,
defrin: multiplier#dma1,
defain: multiplier#xa1,
defrout: multiplier#xb1,
defaout: multiplier#dmb1,
defformat: multiplier#B3, multiplier#B2,
           multiplier#B1, multiplier#B0,
           multiplier#A3, multiplier#A2,
           multiplier#A1, multiplier#A0,
           multiplier#sum7, multiplier#sum6,
           multiplier#sum5, multiplier#sum4,
           multiplier#sum3, multiplier#sum2,
           multiplier#sum1, multiplier#sum0,
defformat: B3, B2, B1, B0, A3, A2, A1, A0,
           Sum7, sum6, Sum5, sum4, Sum3, sum2, Sum1, sum0,
deftest:
xv: 1 1 0 1 1 1 0 1   1 0 1 0 1 0 0 1   ( 13 x 13 = 169 )
xv: 1 0 0 1 1 0 0 1   0 1 0 1 0 0 0 1   (  9 x  9 =  81 )
xv: 1 1 1 1 1 1 1 1   1 1 1 0 0 0 0 1   ( 15 x 15 = 225 )
xv: 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0   (  0 x  0 =   0 )
xv: 0 0 0 1 0 0 0 1   0 0 0 0 0 0 0 1   (  1 x  1 =   1 )
xv: 1 0 0 0 1 0 0 0   0 1 0 0 0 0 0 0   (  8 x  8 =  64 )
xv: 1 1 0 0 1 0 0 1   0 1 1 0 1 1 0 0   ( 12 x  9 = 108 )
xv: 0 1 0 1 0 0 0 1   0 0 0 0 0 1 0 1   (  5 x  1 =   5 )
endtest:
```

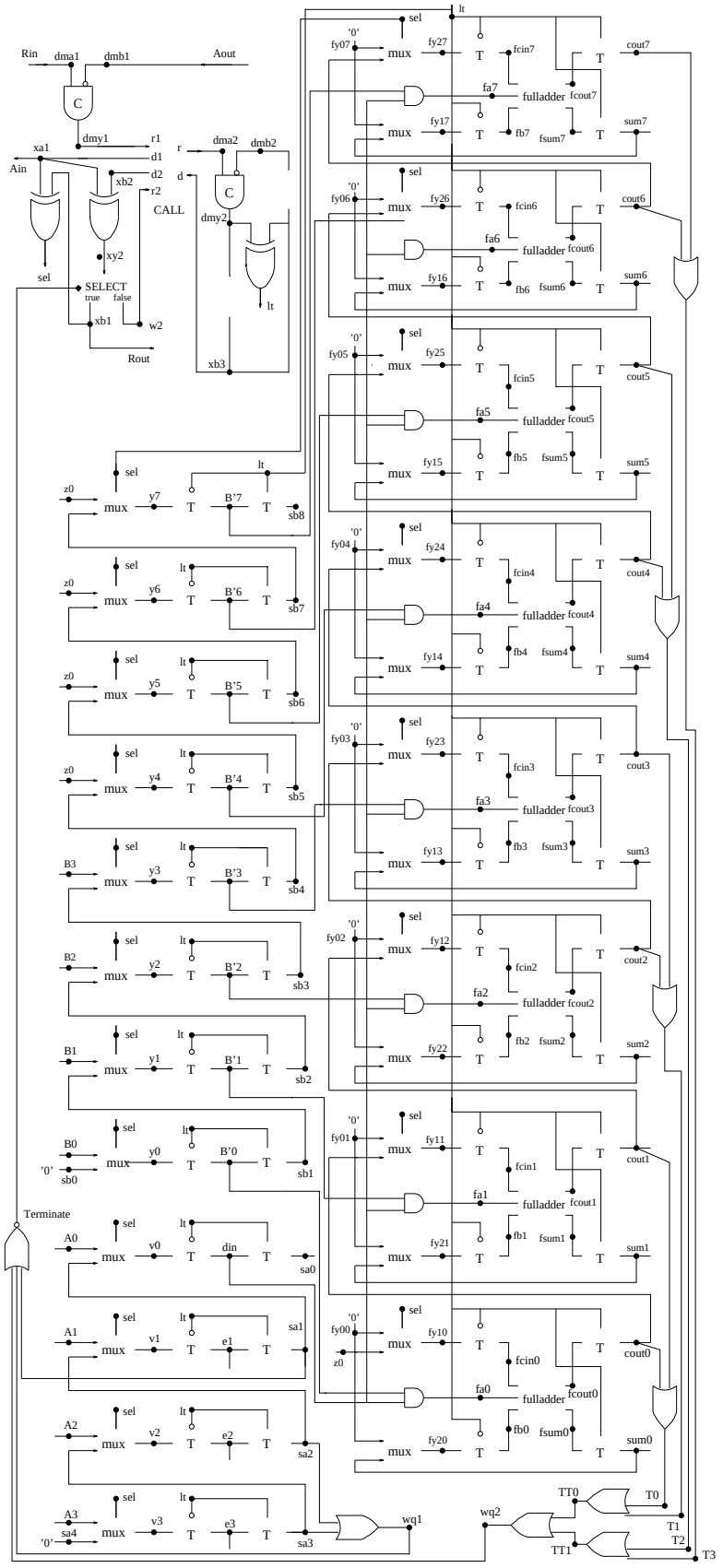


Figure A.9: A four-bit multiplier micropipeline stage

A.6 Two 2-bit multiplier stages joining into one 4-bit multiplier stage

The network whose block diagram is shown in Figure A.10 is also tested. It consists of three micropipeline multiplier stages. They are connected in such a way that two 2-bit micropipeline multiplier stages are joined into one 4-bit micropipeline multiplier stage. The simulation descriptions are shown as follows:

```
definput: m1multiplier#m1B1, m1multiplier#m1B0,
          m1multiplier#m1A1, m1multiplier#m1A0,
          m2multiplier#m2B1, m2multiplier#m2B0,
          m2multiplier#m2A1, m2multiplier#m2A0,
defrin: m1multiplier#m1dma1, m2multiplier#m2dma1,
defain: m1multiplier#m1xa1, m2multiplier#m2xa1,
defoutput: m3multiplier#m3sum7, m3multiplier#m3sum6,
           m3multiplier#m3sum5, m3multiplier#m3sum4,
           m3multiplier#m3sum3, m3multiplier#m3sum2,
           m3multiplier#m3sum1, m3multiplier#m3sum0,
defrout: m3multiplier#m3xb1,
defaout: m3multiplier#m3dmb1,
defformat: m1multiplier#m1B1, m1multiplier#m1B0,
           m1multiplier#m1A1, m1multiplier#m1A0,
           m2multiplier#m2B1, m2multiplier#m2B0,
           m2multiplier#m2A1, m2multiplier#m2A0,
           m3multiplier#m3sum7, m3multiplier#m3sum6,
           m3multiplier#m3sum5, m3multiplier#m3sum4,
           m3multiplier#m3sum3, m3multiplier#m3sum2,
           m3multiplier#m3sum1, m3multiplier#m3sum0,
deftest:
xv: 1 1 0 1 1 1 0 1      0 0 0 0 1 0 0 1
xv: 1 0 0 1 1 0 0 1      0 0 0 0 0 1 0 0
xv: 1 1 1 1 1 1 1 1      0 1 0 1 0 0 0 1
xv: 0 0 0 0 0 0 0 0      0 0 0 0 0 0 0 0
xv: 0 0 0 1 0 0 0 1      0 0 0 0 0 0 0 0
xv: 1 0 0 0 1 0 0 0      0 0 0 0 0 0 0 0
xv: 1 1 0 0 1 0 0 1      0 0 0 0 0 0 0 0
xv: 0 1 0 1 0 0 0 1      0 0 0 0 0 0 0 0
endtest:
```

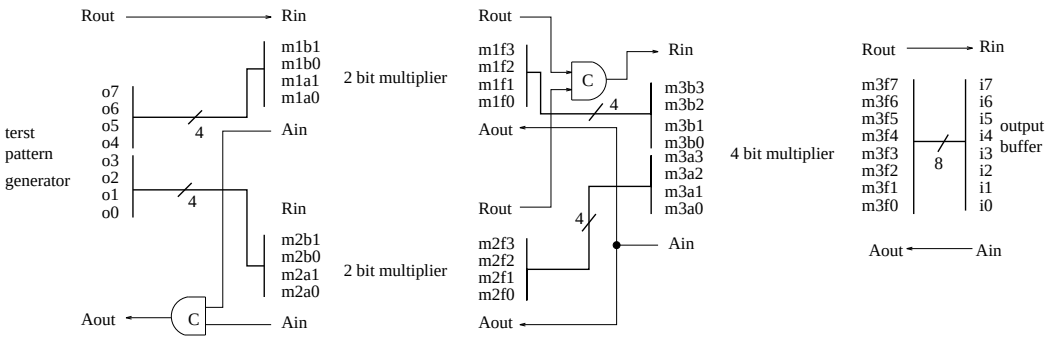


Figure A.10: A complex micropipeline example

After the simulation, the test results are shown as follows:

```
The test results:

                m3f7 m3f6 m3f5 m3f4 m3f3 m3f2 m3f1 m3f0

(3x1) x (3x1)    0    0    0    0    1    0    0    1
(2x1) x (2x1)    0    0    0    0    0    1    0    0
(3x3) x (3x3)    0    1    0    1    0    0    0    1
(0x0) x (0x0)    0    0    0    0    0    0    0    0
(0x1) x (0x1)    0    0    0    0    0    0    0    0
(1x0) x (1x0)    0    0    0    0    0    0    0    0
(3x0) x (2x1)    0    0    0    0    0    0    0    0
(1x1) x (0x1)    0    0    0    0    0    0    0    0
```

Appendix B

Simulating micropipelines using Silos II

Silos II simulator will be used to simulate several micropipeline examples in this appendix. Before starting the simulation a short introduction to Solo 2030 and STL will be given as follows:

Solo 2030

Solo 2030 is a CAD system for VLSI chip designs. It provides an environment for chip design including design entry, simulation, and physical layout. It contains a set of cell libraries (logic gates, I/O cell, ... etc.) and compilers for ROM, RAM, PLA, Multiplier, dual-port RAM, and 2901 datapath megacells, i.e, Solo 2030 has all the features required to handle large complex designs. Solo 2030 includes the industry-standard Silos II and the associated STL simulation and test language. Silos II can be used to simulate a design schematic, generate a waveform from the results, and measure waveform timing characteristics. When simulating a design schematic, a file containing test vectors and a file containing the simulation control are needed. These two files can be generated by STL. More detail about Solo 2030 is given in [European 91].

STL Language

STL (Simulation and Test Language) is a high-level language for the generation of simulation test vectors. It is designed to work with other tools in the Cadence system and is well integrated with the Cadence simulation environment. Let us briefly introduce some STL instructions which will be used in next section. For more details, consult the Solo 2030 Reference Manual.

stlinit initialises STL.

settarget specifies the simulator.

defpin describes pin names, specifies the pin type, such as **in**, **out** or **clk**.

defformat defines the order and the number base of the signals in which the signals appear in the **xv** statement below.

deftiming specifies the resolution of the simulation, the subdivision of a clock cycle, and the duration of a clock cycle.

defstrobe defines the strobe windows to the test equipment for the input and output signals, in terms of the subdivisions of a clock cycle from the deftiming statement.

deftest indicates the start of the simulation.

";" identifies the start of a comment.

xv defines the test vectors.

endtest indicates the end of the simulation.

B.1 Implementing event-driven modules

To simulate the micropipeline circuit shown in Figure 2.4, a dMuller C-element and a TOGGLE are required. The speed of the circuits is not considered in this experiment. Attention is aimed at the function and behaviour of the modules. The event logic modules can be replaced by their high-speed counterparts if such counterparts are available. Therefore, these two event-driven logic modules are simply implemented by some inverters, transparent latches, and XORs. They are described as follows:

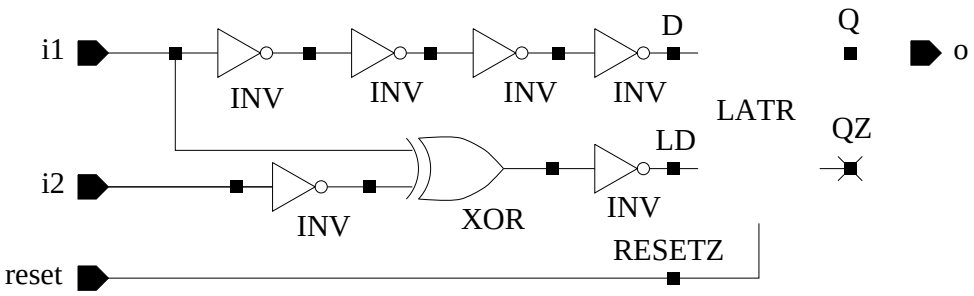


Figure B.1: The implementation of a dMuller C-element

dMuller C-element

Figure B.1 shows a dMuller C-element's schematic which is implemented by six inverters, an XOR, and a high activated transparent latch. The output of the high activated transparent latch is equal to its input if the "LD" is logic "high". The output of the XOR is logic "low" if both its inputs are equal. The inverter connected to the output of the XOR is used to invert the XOR's output. Therefore, the output of the latch is equal to its input if both inputs of XOR are equal. The four inverters are used to delay the "i1" signal, i.e., the "i1" signal should not be faster than the checking signal and the latch will hold the previous state

if both inputs of the XOR are different. The inverter between "i2" and the input of the XOR is to implement the inverted input of the dMuller C-element. This dMuller C-element has been simulated using the following STL program. The simulation result indicates that the function of such dMuller C-element is correct. Its simulation waveform is shown in Figure B.2.

```
stlinit
settarget silos
defpin    reset     in
defpin    i1        in
defpin    i2        in
defpin    o         out
defformat reset i1 i2 o
deftiming 0.01ns 1ns 5ns
defstrobe in    edge     "%...." reset i1 i2
defstrobe out   window   "....%" o
deftest
;         reset  i1  i2  o
xv          1    0   1   0
xv          1    0   0   0
xv          1    1   1   0
xv          1    1   0   1
xv          1    1   1   1
xv          1    0   0   1
xv          1    0   1   0
xv          1    1   0   1
xv          0    0   0   0
xv          0    1   1   0
xv          0    0   1   0
xv          0    1   0   0
endtest
```

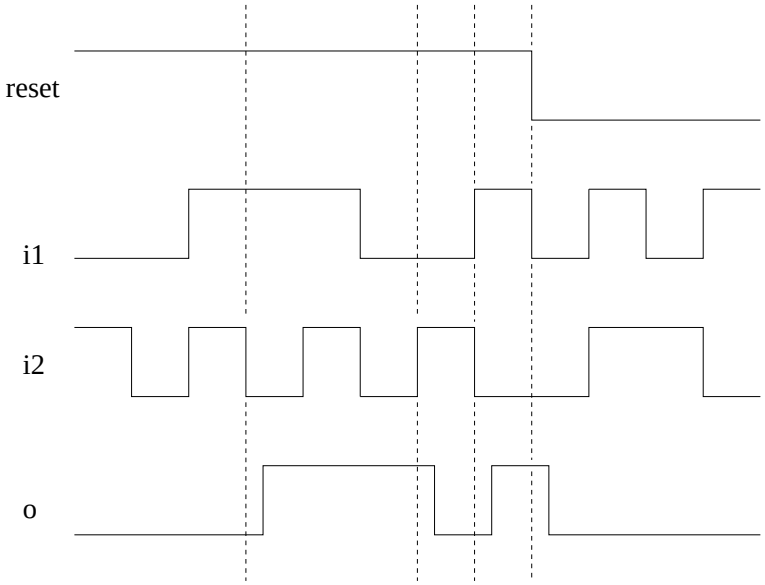


Figure B.2: The simulation waveform of the dMuller C-element

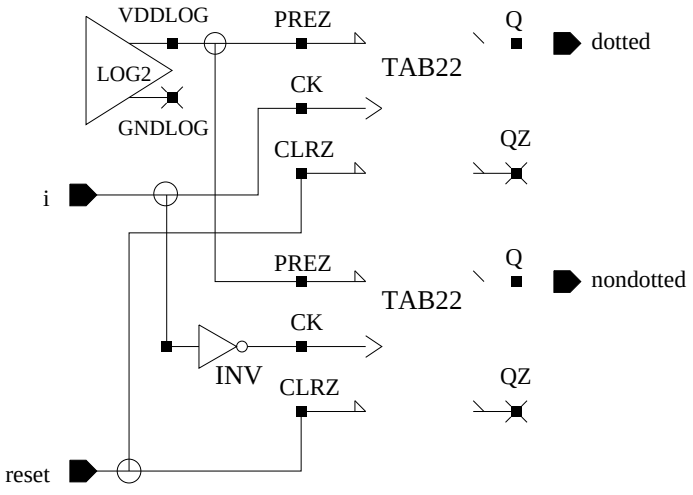


Figure B.3: The implementation of a TOGGLE module

TOGGLE module

Figure B.3 shows a TOGGLE module's schematic which is implemented by one inverter, two edge-trigger T-type FFs (flip-flop), and a logic "high" element. A T-type FF with logic "high" input will change its output state if there is a positive edge appearing on its "CK" input. An inverter connected to the "CK" input of the T-type FF makes it possible that the T-type FF with logic "high" input changes its output state if there is a negative edge appearing on its "CK" input. Therefore, after changing the "reset" signal to logic "high", all rising transitions will change the state of the "dotted" output and all falling transitions will change the state of the "non-dotted" output. This TOGGLE module has been simulated using the following STL program.

```
stlinit
settarget silos
defpin    reset     in
defpin    i         in
defpin    dotted    out
defpin    nondotted out
defformat reset i dotted nondotted
deftiming 0.01ns 1ns 5ns
defstrobe in    edge     "%...." reset i
defstrobe out   window   "....%" dotted nondotted
deftest  ;reset  i dotted nondotted
xv          0    0   0       0
xv          1    0   0       0
xv          1    1   1       0
xv          1    1   1       0
xv          1    0   1       1
xv          1    0   1       1
xv          1    1   0       1
xv          1    1   0       1
xv          1    0   0       0
xv          1    0   0       0
xv          1    1   1       0
xv          1    1   1       0
xv          1    0   1       1
```

```
xv          1    0    1        1
xv          0    1    0        0
xv          0    1    0        0
xv          0    0    0        0
xv          0    0    0        0
xv          0    1    0        0
xv          0    1    0        0
xv          0    0    0        0
xv          0    0    0        0
endtest
```

The simulation result indicates that the function of such TOGGLE module is correct. Its simulation waveform is shown in Figure B.4.

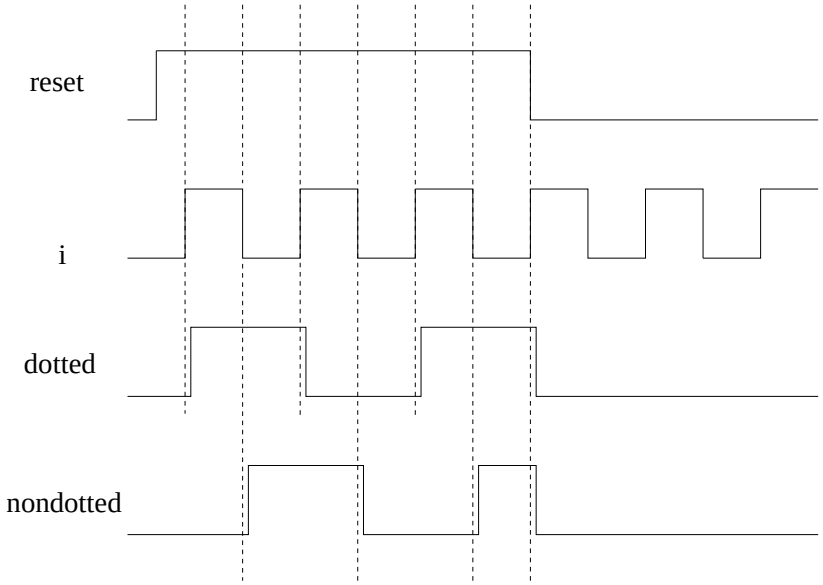


Figure B.4: The simulation waveform of the TOGGLE module

B.2 Simulating micropipelines

A simple micropipeline stage

The schematic of the micropipeline circuit of Figure 2.4 is shown in Figure B.5. After this schematic is entered to the Cadence 2030 system, the following STL program can be used to generate the two simulation files for the Silos II simulator simulating it. The simulation result is shown in Figure B.6.

```
stlinit
settarget silos
defpin    reset      in
defpin    d1         in
defpin    d2         in
defpin    d3         in
defpin    d4         in
defpin    Rin        in
defpin    Aout       in
defpin    Rout       out
defpin    Ain        out
defpin    dout       out
defformat reset d1 d2 d3 d4 Rin Aout Rout Ain dout
deftiming 0.01ns 1ns 5ns
defstrobe in    edge     "%...." reset d1 d2 d3 d4 Rin Aout
defstrobe out   window   "....%" Rout Ain dout
deftest
;         reset  d1 d2 d3 d4 Rin Aout Rout Ain dout
xv          0    0   0  0  0   0  0      X  X  X
xv          1    1   1  1  1   0  0      X  X  X
xv          1    1   1  1  1   1  0      X  X  X
xv          1    1   1  1  1   1  1      X  X  X
xv          1    1   0  1  0   1  1      X  X  X
xv          1    1   0  1  0   0  1      X  X  X
xv          1    1   0  1  0   0  0      X  X  X
endtest
```

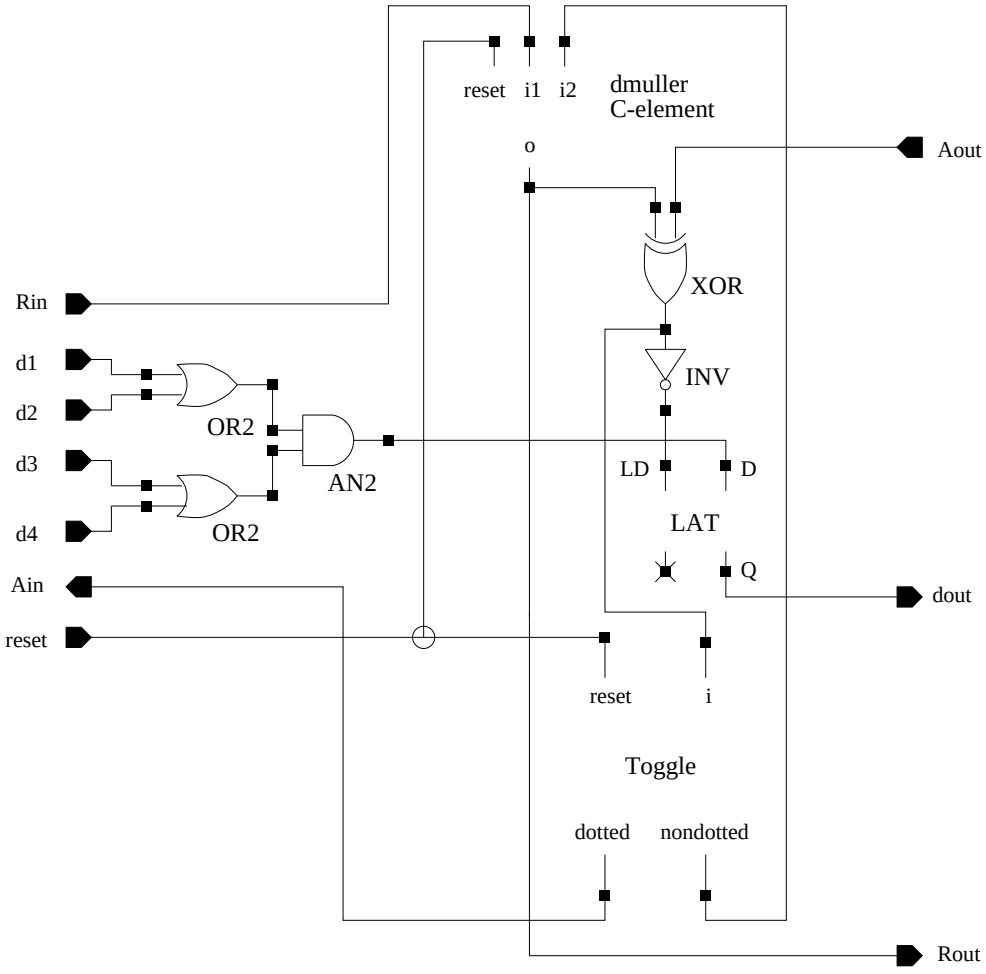


Figure B.5: The schematic of the micropipeline circuit of Figure 2.4

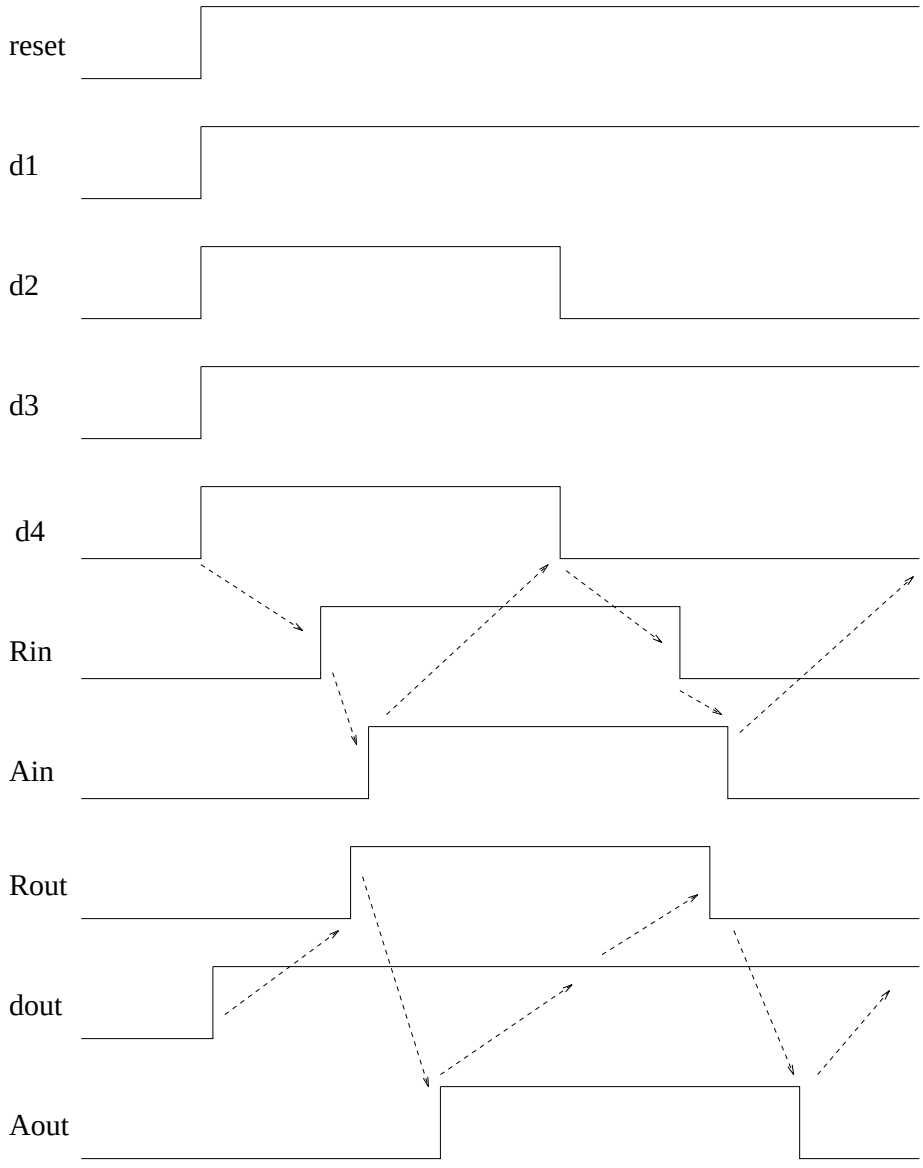


Figure B.6: The simulation waveform of the schematic of Figure B.5

The schematic of another micropipeline stage whose combinational logic circuit consists of one inverter is shown is Figure B.7. It can be simulated using the following STL program and the simulation result is shown in Figure B.8.

```
stlinit
settarget silos
defpin    reset      in
defpin    d1         in
defpin    Rin        in
defpin    Aout       in
defpin    Rout       out
defpin    Ain        out
defpin    dout       out
defformat reset d1 Rin Aout Rout Ain dout
deftiming 0.01ns 1ns 5ns
defstrobe in    edge     "%...." reset d1 Rin Aout
defstrobe out   window   "....%" Rout Ain dout
deftest
;         reset  d1 Rin Aout Rout Ain dout
xv          0    0   0   0     X   X   X
xv          1    1   0   0     X   X   X
xv          1    1   1   0     X   X   X
xv          1    1   1   1     X   X   X
xv          1    0   1   1     X   X   X
xv          1    0   0   1     X   X   X
xv          1    0   0   0     X   X   X
endtest
```

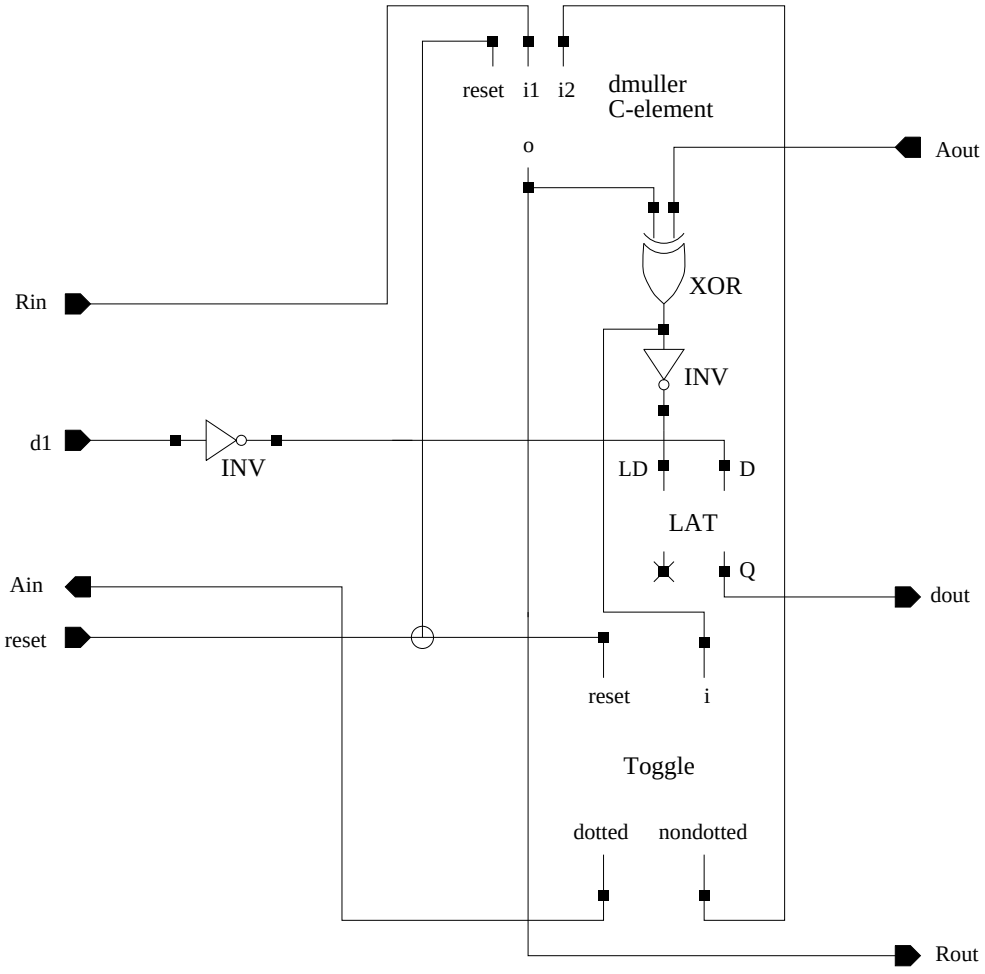


Figure B.7: The schematic of the a invert micropipeline circuit

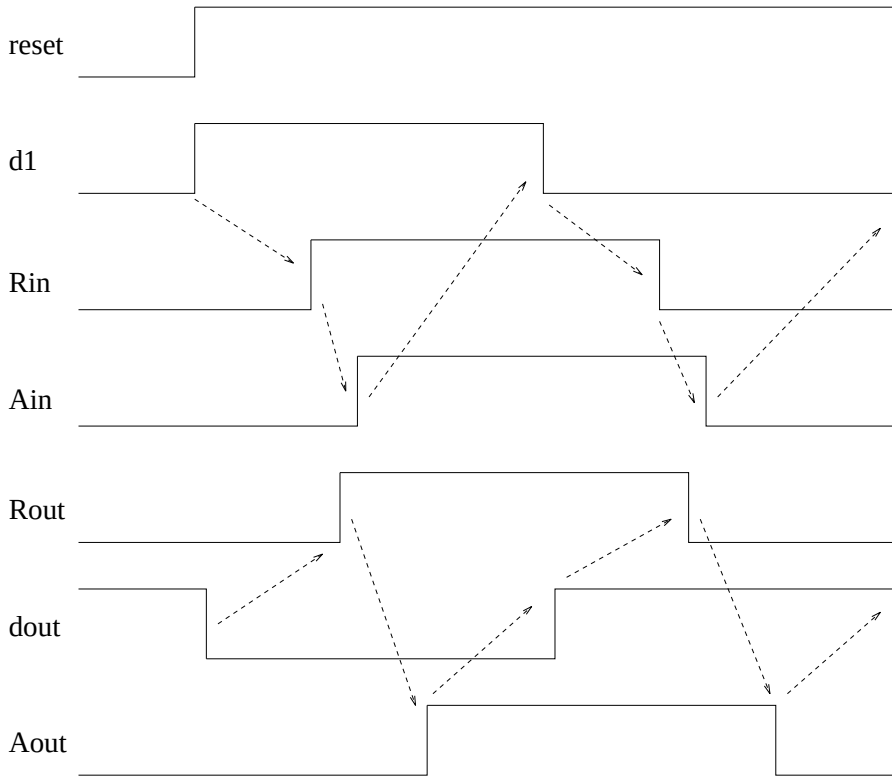


Figure B.8: The simulation waveform of Figure B.7

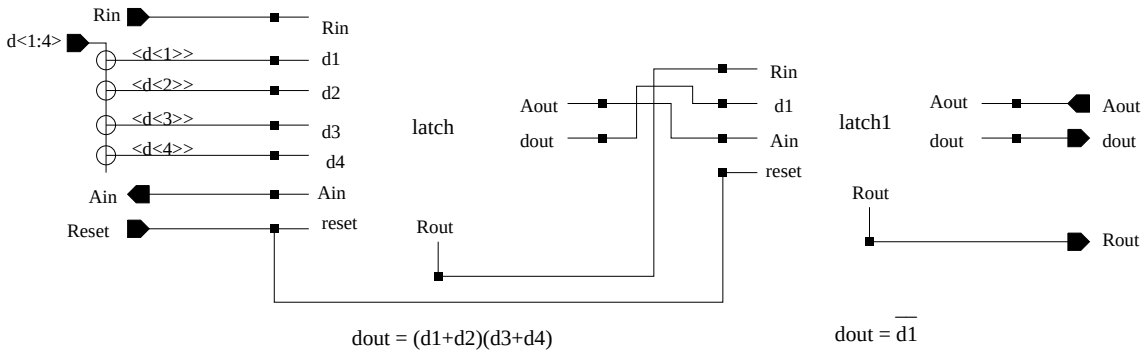


Figure B.9: The schematic of two serially connected micropipeline stages

Two serially connected micropipeline stages

The schematic of two micropipeline stages of Figure A.1 is shown in Figure B.9. It consists of the circuits of Figures B.5 and B.7. The following STL program can be used to simulate it and the simulation result is shown in Figure B.10.

```
stlinit
settarget silos
defpin    reset      in
defpin    d<1:4>     in
defpin    Rin        in
defpin    Aout       in
defpin    Rout       out
defpin    Ain        out
defpin    dout       out
defformat reset d:hex Rin Aout Rout Ain dout
deftiming 0.01ns 1ns 5ns
defstrobe in    edge     "%...." reset d Rin Aout
defstrobe out   window   "....%" Rout Ain dout
deftest
;         reset  d<1:4> Rin Aout Rout Ain dout
xv          0    0x0     0   0    X    X   X
xv          1    0xF     0   0    X    X   X
xv          1    0xF     1   0    X    X   X
xv          1    0xF     1   1    X    X   X
xv          1    0x3     1   1    X    X   X
xv          1    0x3     0   1    X    X   X
xv          1    0x3     0   0    X    X   X
endtest
```

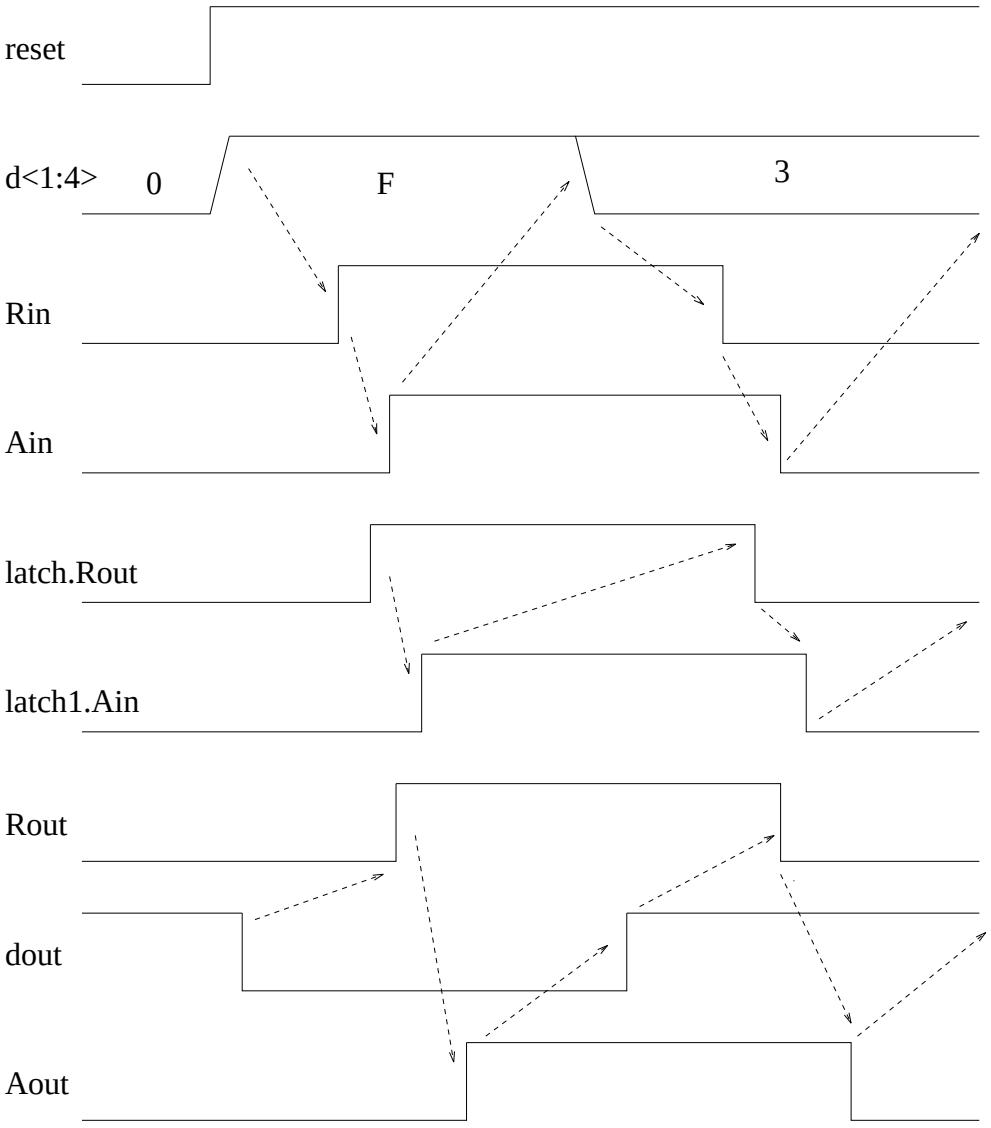


Figure B.10: The simulation waveform of Figure B.9

Bibliography

[Agerwala 79] Agerwala Tilak.: Special Feature: Putting Petri Nets to Work, IEEE Computer, Vol. 12, No. 12, Dec. 1979, pp. 85-94.

[Brunvand 89] Brunvand, Erik and Sproull, F. Robert: Translating Concurrent Communicating Programs into Delay-Insensitive Circuits, Carnegie Mellon University, Apr. 1989, Technical Report CMU-CS-89-126.

[Calvo 86] Calvo, J. and Acha, J.L. and Valencia, M.: Asynchronous Modular Arbiter, IEEE Transactions on Computers, Vol. c-35, No. 1, Jan. 1986, pp. 67-70.

[Chu 86] Chu, Tam-Anh: On the Models for Designing VLSI Asynchronous Digital Circuits, INTEGRATION, the VLSI Journal, Vol. 4, No. 2, Jun. 1986, pp. 99-113.

[Day 91] Day, Paul: A 4-Bit Asynchronous Multiplier Circuit, Draft copy, AMULETT Group, Department of Computer Science, University of Manchester, Apr. 1991.

[Dill 92] Dill, David L. and Nowick, Steven M. and Sproull, Robert F.: Specification and Automatic Verification of Self-Timed Queues, Formal Methods in System Design, Vol. 1, Jul. 1992, pp. 29-60, Kluwer Academic Publishers.

[European 91] Solo 2030 User Guide, European Silicon Structures, 1991.

[Ellis 90] Ellis, Margaret A. and Stroustrup, Bjarne: The Annotated C++ Reference Manual, Addison-Wesley, 1990.

[Ginosar 90] Ginosar, Ran and Michell, Nick: On the Potential of Asynchronous Pipelined Processors, Computer Architecture News, Vol. 18, No. 4, Dec. 1990, pp. 27-34.

[Gopalakrishnan 90] Gopalakrishnan, Ganesh and Jain, Prabhat: Some Recent Asynchronous System Design Methodologies, Department of Computer Science, University of Utah, Oct. 1990, Technical Report UU-CS-TR-90-016.

[Greenstreet 88] Greenstreet, Mark R. and Steiglitz, Kenneth: Throughput of Long Self-Timed Pipelines, Department of Computer Science, Princeton University, Technical report CS-TR-190-88, Nov. 1988.

[Hennessy 90] Hennessy, John L. and Patterson, David A.: Computer Architecture – A Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

[Joerg 90] Joerg, Werner B.: A subclass of Petri nets as design abstraction for parallel architectures, Computer Architecture News, Vol. 18, No. 4, Dec. 1990, pp. 67-77.

[Komori 88] Komori, Shinji and Takata, Hidehiro and Tamura, Toshiyuki and Asai, Fumiyasu and Ohno, Takio and Tomisawa, Osamu and Yamasaki, Tetsuo and Shima, Kenji and Asada, Katsuhiko and Terada, Hiroaki: An Elastic Pipeline Mechanism by Self-Timed Circuits, IEEE Journal of Solid-State

Circuits, Vol. 23, No. 1, Feb. 1988, pp. 111-117.

[Lavagno 91] Lavagno, L. and Keutzer, K. and Sangiovanni-Vincentelli, A.: Algorithms for Synthesis of Hazard-free Asynchronous Circuits, 28th ACM/IEEE Design Automation Conference, 1991, pp. 302-308.

[Lippman 90] Lippman, Stanley B.: C++ Primer, Addison-Wesley, 1990.

[Martin 89] Martin, J. Alain and Burns, Steven M. and Lee, T. K. and Borkovic, Drazen and Hazewindus, Pieter J.: The First Asynchronous Microprocessor: The Test Results, Computer Architecture News, Apr. 1989.

[Martin 91] Martin, Alain J.: Synthesis of Asynchronous VLSI Circuits, Department of Computer Science, California Institute of Technology, Aug. 1991.

[Martin 92] Martin, Alain J.: Asynchronous Datapaths and the Design of an Asynchronous Adder, Formal Methods in System Design, Vol. 1, Jul. 1992, pp. 117-137, Kluwer Academic Publishers.

[Meng 89] Meng, Teresa H.-Y. and Brodersen, Robert W. and Messerschmitt, David G.: Automatic Synthesis of Asynchronous Circuits from High-Level Specifications, IEEE Transactions on Computer-Aided Design, Vol. 8, No. 11, Nov. 1989, pp. 1185-1205.

[Miller 65] Miller, Raymond E.: Switching Theory Vol. II: Sequential Circuits and Machines, John Wiley, 1965.

[Misunas 73] Misunas, David: Petri Nets and Speed Independent Design, Communications of the ACM, Vol. 16, No. 8, Aug. 1973, pp. 474-481.

[Pearce 75] Pearce, R.C. and Field, J.A. and Little, W. D.: Asynchronous Arbiter Module, IEEE Transactions on Computers, Vol. c-24, Sep. 1975, pp. 931-932.

[Peterson 77] Peterson, James L.: Petri Nets, Computing Surveys, Vol. 9, No. 3, Sep. 1977, pp. 223-252.

[Peterson 81] Peterson, James L.: Petri Net Theory and the Modeling of Systems, 1981, Prentice-Hall.

[Plummer 72] Plummer, William W.: Asynchronous Arbiters, IEEE Transactions on Computers, Vol. c-21, Jan. 1972, pp. 37-42.

[Ramamoorthy 80] Ramamoorthy, C.V. and Ho, Gary S.: Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets, IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, Sep. 1980, pp. 440-449.

[Seitz 80] Seitz, Charles L.: System Timing, In "Introduction to VLSI Systems" edited by Carver Mead and Lynn Conway, Addison-Wesley, 1980.

[Stroustrup 91] Stroustrup, Bjarne: The C++ Programming Language, Second Edition, Addison-Wesley, 1991.

[Sutherland 89] Sutherland, E. Ivan: Micropipelines, Communications of the ACM, Vol. 32, No. 6. Jun. 1989, pp. 720-738.

[Unger 69] Unger, Stephen H.: Asynchronous Sequential Switching Circuits, Wiley-Interscience, 1969.

[Williams 90] Williams, Ted: Latency and Throughput Tradeoffs in Self-Timed Speed-Independent Pipelines and Rings, Computer Systems Laboratory, Stanford University, Aug. 1990, Technical Report CSL-TR-90-431.