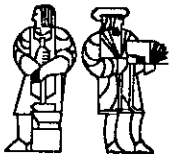
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

The Design, Implementation and Testing of a Self-timed Two by Tow Packet Router

Computation Structures Group Memo 236
February 1983

Tam-Anh Chu

This research was supported by the National Science Foundation under grant MCS-7915255.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

The Design, Implementation and Testing of a Self-Timed Two by Two Packet Router[1]

Tam-Anh Chu
MIT Laboratory for Computer Science
Cambridge Massachusetts 02139

Abstract

This report presents the design, implementation and testing of a two by two packet router. An experimental version of the router has been fabricated as a NMOS LSI chip. A thorough discussion of a self-timed design methodology is carried out, including some basic definitions and classification of self-timed systems. The design and layout of the router are then presented in detail. A test strategy is introduced, which would allow testing of all multiple stuck at faults in the system at constant cost. An evaluation of the design methodology shows that despite of the relatively large amount of area required for layout, it provides many advantages over other conventional design approaches in terms of speed, ease of design and modification, and lastly, verification and testing.

---

1. This research was supported by the National Science Foundation under grant number MCS-7915255.

CONTENTS

1. Introduction

This report describes the design, implementation and testing of a NMOS LSI router chip, using a design methodology different from other conventional ones, namely, the self-timed design methodology. The reason for investigating this design methodology is twofold. First, as has been pointed out in [17], as the basic dimension of the devices on an IC chip is scaled down, the rate of signal transmission in the interconnects would scale up quadratically , whereas the switching time of the devices scales down linearly. Thus the ratio of signal transmission time to gate delay would be increased as the cubic power of the scaling factor. In a synchronous system with scaled down dimensions, then, most of the available time has to be allocated for signal transmission, and the real data processing will take up only a small fraction. Moreover, additional time must be allowed to deskew global signals reaching different parts of the system, and this will also consume a part of the total time. If a synchronous system is to be built this way, it will be very inefficient. These problems may become unmanageable as the system gets more complex, and a feasible solution is to partition the system into synchronous blocks communicating with each other asynchronously. However, such a scheme would necessitate the use of a communication protocol between these blocks and a mechanism for synchronizing asynchronous signals. Unfortunately, it has been shown that all synchronizers can be induced into the meta-stable state by coincident asynchronous inputs [3]. In the meta-stable state, the output of the synchronizer is not correct and valid, and therefore will be interpreted inconsistently by different parts of the system. Self-timed systems, on the other hand, use the asynchronous communication protocol as a means of transferring data between modules; also, since no clock is required, meta-stability is not a problem.

Another motivation for using self-timed design is its direct and straightforward implementation of packet communication systems [7], being systems composed of modules communicating with each other by sending data packets to each other using an asynchronous communication protocol.

This report is organized as follows. Section 2 gives the background information about a particular self-timed design method, including a definition of self-timed circuits and the assumptions on gate and wire delays. The implementation of basic self-timed modules is presented in Section 3. Section 4 presents a top down design approach for self-timed systems and its apllication to the design of a router, and information about the physical chip including the layout and floor plan. Section 5 discusses a test strategy for the router, and a test circuit is described. Section 6 evaluates the design methodology based on the projected area, power consumption and speed of the IC, related issues

such as optimization, design modification and verification are considered. Section 7 summarizes the main results and suggests directions for further research.

2. Basic Theory

In this section, we attempt to introduce a number of fundamental notions related to self-timed design. First, it is necessary to have a definition of self-timed circuits and to state a number of assumptions on the properties of circuit elements used to realize them. Then, we will discuss methods of implementing self-timed modules, including combinational logic modules, state machines, and other determinate and non-determinate ones.

2.1 Definitions

A self-timed system can be defined as a system consisting of circuit modules, communicating with each other locally, without using any global reference signal. The circuit modules "time" the processing and transmission of data by themselves, and the sequence of operations is indicated by the start and completion of modules instead of by clock ticks. In this sense, we can consider self-timed circuit as a subclass of asynchronous digital circuits, being different from the conventional ones in that it imposes a uniform communication protocol on all system modules, whereas the latter do not have a systematic restriction on how signals change and interact in the system. Experiences demonstrated that conventional asynchronous design is at best difficult and therefore can only be applied to small designs. On the contrary, by enforcing a global discipline, self-timed design is reasonably straightforward : large and complex systems can be implemented with ease by composition of smaller components.

A closely related notion to self-timed system design is *speed-independence* or *delay-insensitiveness*, meaning that the correct operation of a system is not affected by the variation of delays of logic gates and wires. Usually, in most designs, wire delays are assumed negligible, and this assumption would lead to a particular implementation of self-timed modules. The importance of the basic assumptions on gate and wire delays will be discussed later in this section. One should note that a self-timed system could be designed to be completely operational without being speed-independent, therefore, speed-independence can be considered a highly desirable, but not an essential property of self-timed systems.

2.2 Assumptions on Gate and Wire Delays

As mentioned earlier, gate and wire delays assumptions are fundamental to the design and implementation of self-timed systems. A statement of the models used for gate and wire delays is required before one can proceed to the implementation stage. First, we would like to discern between two delay models.

A *pure delay* [21] D can be visualized as a lumped network with one input and one output. If the delay is excited by a pulse whose width is larger than the time duration D, then a delayed version of the pulse will be propagated to the output, otherwise, it will be suppressed completely. Pure delays can be used to model logic gate delays with reasonable accuracy.

A *stray delay* is due to wires' parasitic resistance and capacitance, it delays the input signals and distorts their shapes. This type of delay reflects the property of interconnects. Since signal transmission in wires is basically an analog process, a simplified linear model usually will not be sufficiently accurate. For example, the stray delays do not add up linearly, as opposed to pure delays. To avoid this difficulty, we will in general neglect wire delays if wire lengths are sufficiently short.

One can assume that gate/wire delays are either bounded or unbounded. The unboundedness of a delay should be interpreted as the inability to characterize a delay and place a bound on it, eventhough it may be finite. Traditionally, this has been true : in the early days, logic gates were built from discreet components with such a wide range of parameter variations that it was almost impossible to characterize each component accurately. The term *time dispersion* had been appropriately used to describe the property of these badly timed elements [12].

The particular set of self-timed modules described later is designed using the following assumptions :

1. Logic gates have unbounded delays.
2. Wires within a module have negligible delays.
3. Wires connecting modules can have unbounded delays.

Other methods of implementation of self-timed systems can be classified according to the assumptions on gate and wire delays. If a gate is defined as a basic logic element built from transistors and not containing any feedback, then we can identify the following three classes, as shown in Figure 1.

1. In the first class, both gates and wires are assumed to have unbounded delays. This type of design was proposed by Muller [14]. The synthesized modules are self-timed, speed independent and pipelined. It was assumed that perfect C-elements (see section 3.3) were available, i.e., they were basic logic elements and not synthesized from logic gates with feedback. Dual rail coded signals were used to transmit data between modules, this would ensure correct operation under the unbounded wire delays assumption.

2. In the second class, gates are assumed to have unbounded delays, whereas wires negligible delays. Self-timed circuits synthesized using Petri nets fit under this class and are speed-independent. An implementation of the control logic for the CDC 6600 and of some data logic including registers/counters have been reported in [6, 8], Petri nets realization using logic circuits and asynchronous logic arrays in [15, 16], Petri nets modeling of self-timed circuits in [13].

3. The third class of self-timed circuits implementation assumes logic gates with bounded delays and wires with negligible delays. Circuits realizable under this class is not speed independence. Some examples are the TriMosBus reported in [20], and a design of the FIFO found in [18].

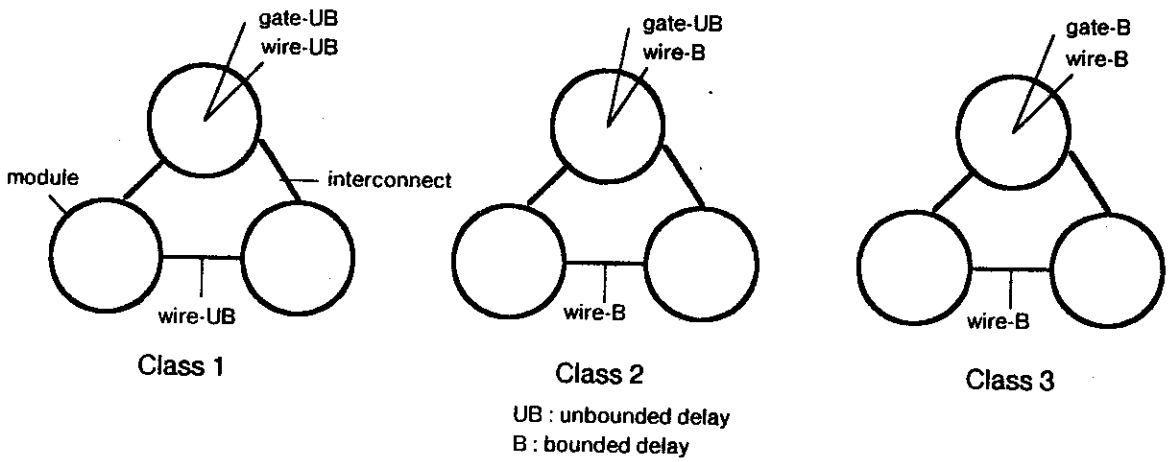


Figure 1. Classes of Self-timed Design

The above classification is somewhat artificial in that the division depends completely on how one defines a basic logic element. If the C-element were defined as such a basic element, then the division would change. However, we have chosen to define the logic gate with no feedback as the basic element, and this allows us to categorize and discuss about these classes more systematically. Even though this report focuses on one class of implementation only, this classification emphasizes the importance of assumptions on gate/wire delays.

3. Implementation of Self-timed Systems

A self-timed system is a network of interconnected self-timed modules, each of them has a number of characteristics, as described below.

3.1 General Characteristics of Self-timed Modules

1. From the above definition of self-timed circuits, a closed loop communication between modules is found necessary. This is accomplished through the use of an asynchronous signaling protocol with *ready* and *acknowledge* signals, used to signal the transfer of data between modules. We propose the use of the reset signaling protocol, in which a signaling cycle contains an active phase and a reset phase as shown in Figure 2. This signaling protocol is preferred over the two-cycle one as it requires less circuitry to implement, and also allows system initialization to be done more easily.

2. For a self-timed module, the relationship between the input and output data is strictly causal. In the active phase of the signaling cycle, input data have to be completely defined and stable before the output data are; in the reset phase, input data have to reset to an idle state before the output data. Such strictly causal relationships are termed the weak-conditions in [18] and shown in Figure 3. As discussed later, they can be violated during the reset phase without affecting the operation of the system. The reason for this flexibility is that the weak-conditions constitute only a part of the

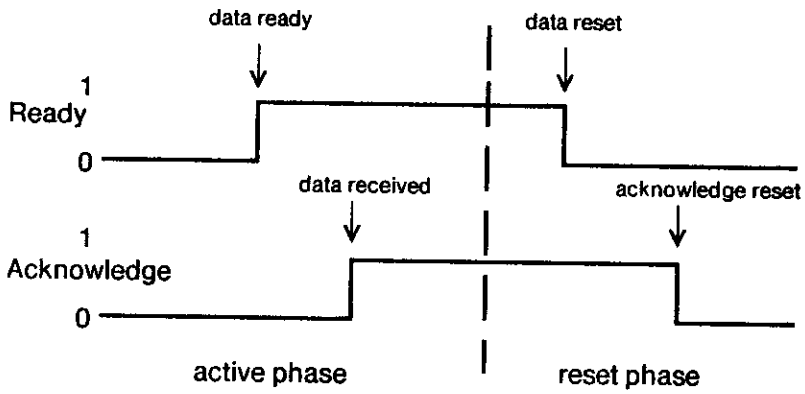


Figure 2. The Reset Signaling Protocol

specification of the communication protocol. A complete protocol also needs to specify the relation between the data and the *ready/acknowledge* signals at the input and output ports of the modules.

3. By coupling the temporal relations between data signals and *ready/acknowledge (R/A)* signals at the input and output ports, the complete terminal behavior of a self-timed module can be described. At each input or output port, *R/A* signals are related to data signals as shown in Figure 4. A module can be a pipelined or non-pipelined one, depending on the relation between the input *R/A*

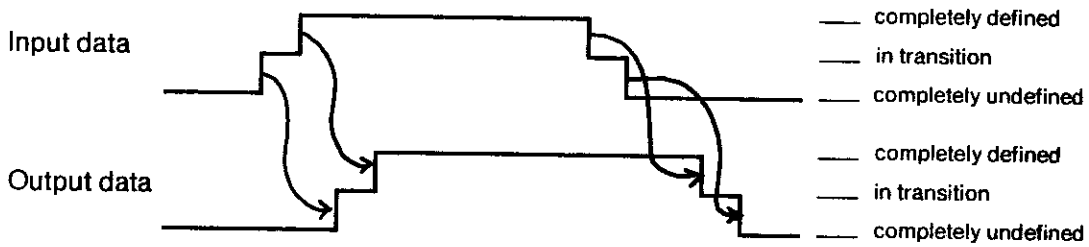


Figure 3. Input/Output Relation of a Self-timed Module

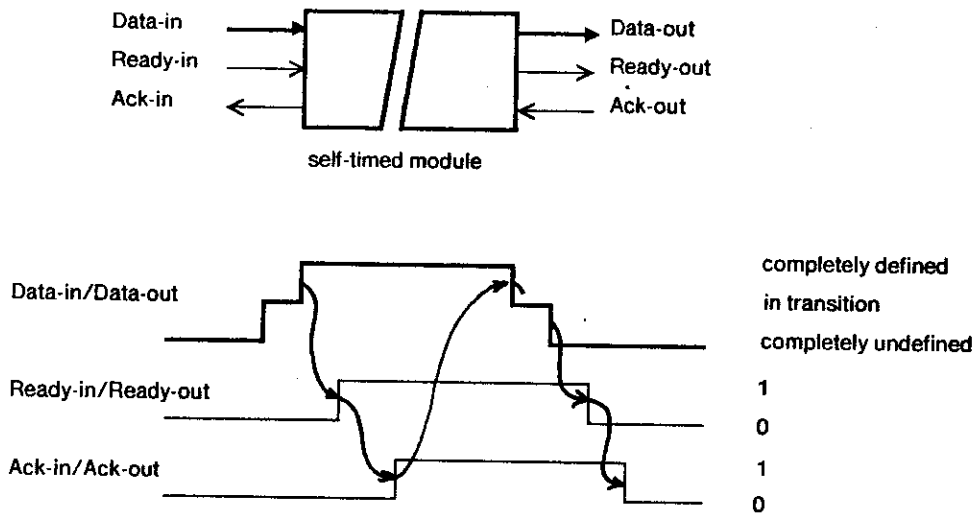


Figure 4. Relation between Data and R/A Signals in the Signaling Protocol

and output *R/A* signals, as shown in Figure 5. It also shows the precedence graphs for both cases, where more parallelism can be detected in the graph associated with the pipelined modules. A non-pipelined module merely propagates the *R/A* signals through to other modules, whereas a pipelined one actually returns the *R/A* signals as soon as it could so that subsequent data can be sent immediately. There are many uses of a pipelined module : it can pipeline the system operation, form feedback loops in state machines, and lastly, it can be used for resynchronizing the timing relation between the *R/A* signals and the data in case explicit *R/A* signals are used.

4. Dual rail coded signals are used to transmit data between modules. In the dual rail format, a data signal x will be coded in two wires x_1, x_0. For x equal to 0 and 1, x_1 and x_0 are coded with data states 01 and 10, respectively. The 00 state corresponds to the spacer, where no signal is transmitted; state 11 is prohibited. At each port of a module, data and spacer states have to alternate, as shown in Figure 6. The use of dual rail code is much more fundamental than just an implementation detail, as it has direct consequences on the size and design of modules, the correctness of system operation, and on fault modeling and testing of self-timed systems. These implications will be explored in later sections. One immediate consequence of the use of dual rail code is that an explicit *ready* signal is no longer required, and signal transmission between modules is

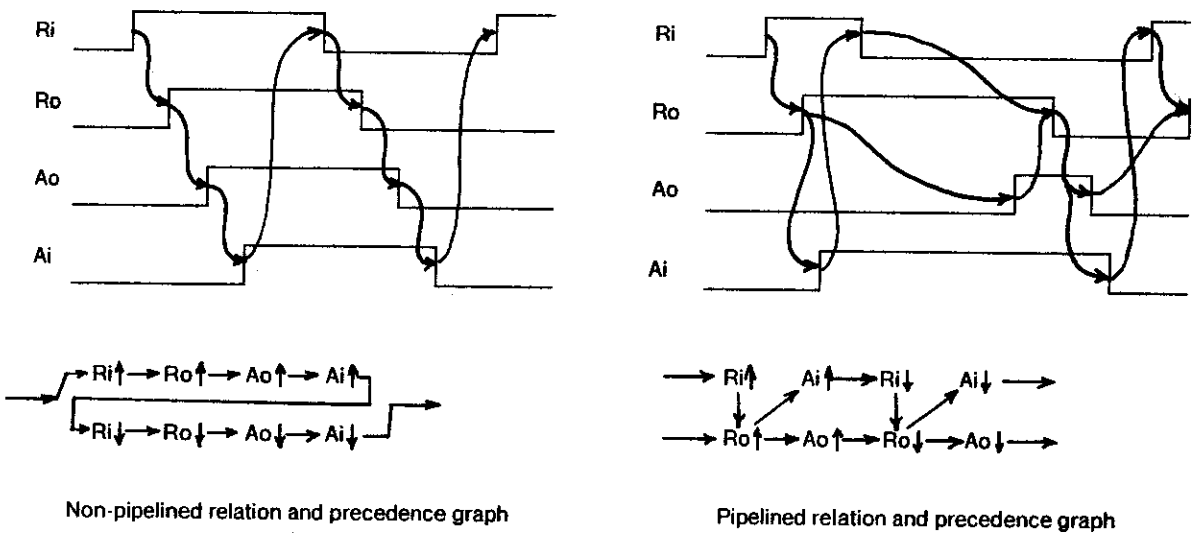


Figure 5. Non-Pipelined and Pipelined Relations

insensitive to wire delays.

3.2 Construction of Self-timed Systems

Arbitrary self-timed systems can be constructed from modules with the above characteristics. The correctness of the a system is guaranteed if the following rules for construction are observed :

1. The input/output links of modules match each other (A link consists of data and *R/A* signal wires).

2. No links are left dangling, i.e., every port has to be terminated by other modules or at the input/output of the system.

3. The system configuration or connection does not cause deadlock. For example, a non-pipelined module with an output link connected to one of its input links will cause deadlock.

It can be proved easily by induction that systems constructed from self-timed modules following these construction rules are also self-timed. One can draw an arbitrary boundary around a number of modules to obtain a new self-timed module. We define a primary module is a self-timed module which can not be decomposed further to other self-timed modules; a secondary module is one consisting of more than one primary one.

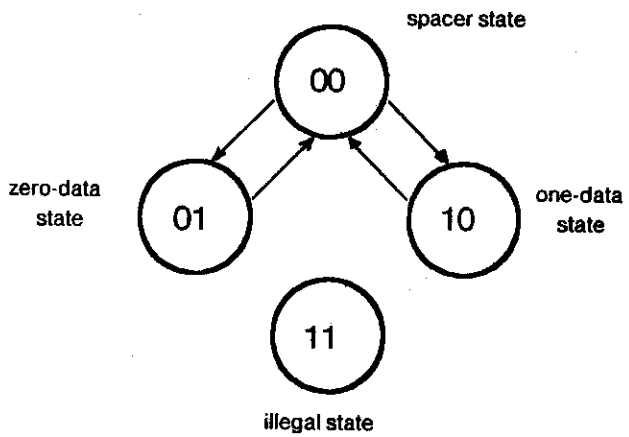


Figure 6. The Dual Rail Code

3.3 Synthesis of Self-timed Modules

In this section, we describe a method for synthesizing self-timed modules from elementary logic gates such as ANDs, ORs, etc.. The C-element and the arbiter circuit, being two critical and basic building blocks will also be described. Determinate, nondeterminate and non well-formed modules will be then introduced to complete the set of self-timed modules.

3.3.1 Basic Building Blocks

The implementation of self-timed modules requires the use of common logic gates such as ANDs, ORs, and also Muller's C-element and the arbiter circuit. The symbol and logic circuit diagram of the C-element are shown in Figure 7. It is an asynchronous state machine whose output changes only when two inputs are the same but different from the current output state. As discussed in [5], the correct operation of the system is guaranteed if the C-element is designed such that its loop delay is less than the time interval between two input transitions in opposite directions. Another important element is the arbiter circuit, which consists of a cross-coupled NOR gates front-end and a threshold detection circuit, as shown in Figure 8. The threshold circuit will suppress the illegal voltage level associated with the bistable device when it is in the metastable state. The metastable state behavior and the design of the arbiter circuit have been analyzed in [5].

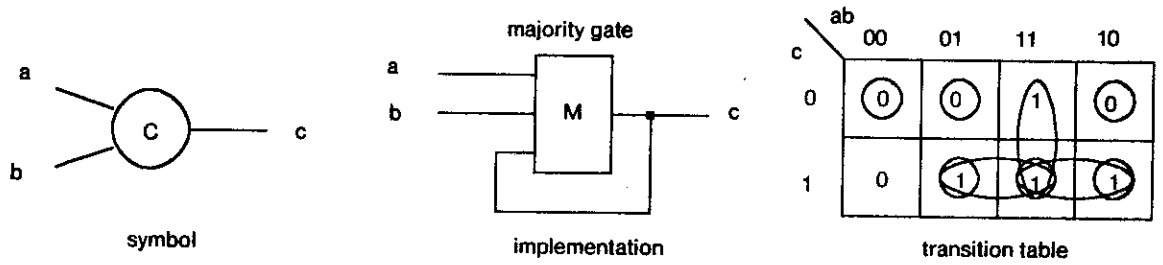


Figure 7. C-element

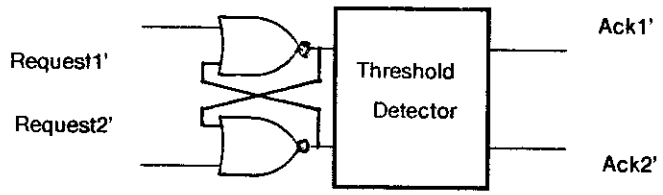


Figure 8. The Arbiter Circuit

3.3.2 Combinational Logic Modules

Realization of combinational logic for self-timed circuits under the unbounded gate delay assumption is unique in that it is different from both the conventional synchronous and asynchronous methods. For logic gates with unbounded delays, a particular logic implementation can be free of static and dynamic hazards but a new type of hazard, called delay hazard, can still exist in the circuit [1]. If one assumes a combinational logic function f and lets I_1, I_2, I_3 be three consecutive input states to the function f, then static and dynamic delay hazards can be defined as

> Static – Input sequence $I_1I_2I_3$ yields output sequence $f(I_1)f(I_2)f'(I_2)f(I_3)$, where $f(I_2) = f(I_3)$.
>
> Dynamic – Input sequence $I_1I_2I_3$ yields output sequence $f(I_1)f(I_2)f(I_3)f(I_2)f(I_3)...$, where $f(I_2) \neq f(I_3)$.

Figure 9 show examples of static and dynamic delay hazards. A common technique for eliminating delay hazards in combinational logic circuits is to code the data in a pure code, such as the m out of n (*m/n*) code. Each code word is n bit long and a valid code contains m 1's, where m is called the weight. The dual rail code is such a pure code, namely the *1/2* code. The *1/2* code is chosen instead of other *m/n* codes with *m* greater than $\frac{1}{2}n$ because for the *1/2* code, encoding and decoding circuits are simple, and the amount of circuitry required is linearly proportional to the fan-in or fan-out of the modules.

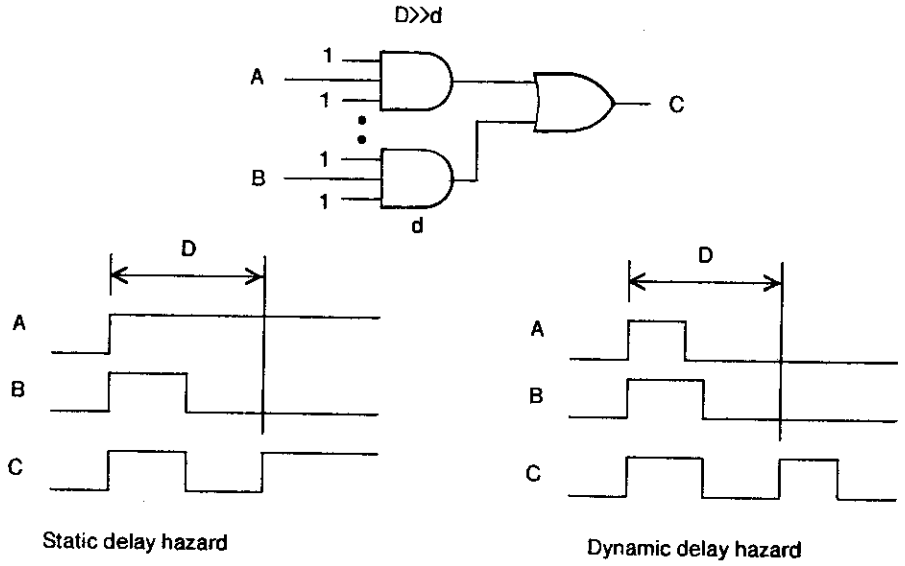


Figure 9. Examples of Static and Dynamic Delay Hazards

In using this code to implement combinational logic, we require that data alternate between a valid code word and a spacer (corresponding to the all zero word). In order to avoid any type of hazard, the logic function has to map any input word with weight less than m to an output spacer, and any valid input code word to a valid output code word. It is assumed that the code words with weight greater than m never occur, and these would correspond to don't care conditions in the logic realization. This mapping process is depicted graphically in Figure 10.

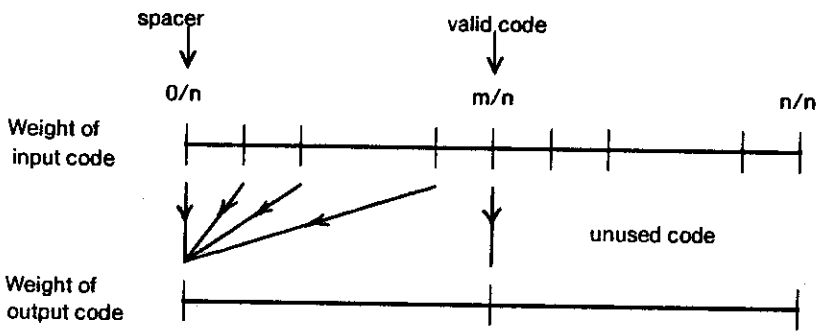


Figure 10. Input/Output Mapping for 1/2 Code

For dual rail code, the logic equation for a hazard free implementation of a function can simply be extracted from the Karnaugh's map representation of the original uncoded function, where the 1-signal of the coded function is the sum of all minterms corresponding to the 1's in the map, the 0-signal the sum of all minterms corresponding to the 0's in the map. For example, the logic equations for the dual rail coded AND function are

$$x_0 = a_0.b_0 + a_0.b_1 + b_1.a_0$$
$$x_1 = a_1.b_1$$

Hence, the dual rail implementation of a function $y = F(x_1,x_2,...,x_n)$ requires 2^n minterms, each is a unique vector belonging to the interval

$$\langle x_1x_2...x_n\ ,\ x'_1x'_2...x'_n \rangle$$

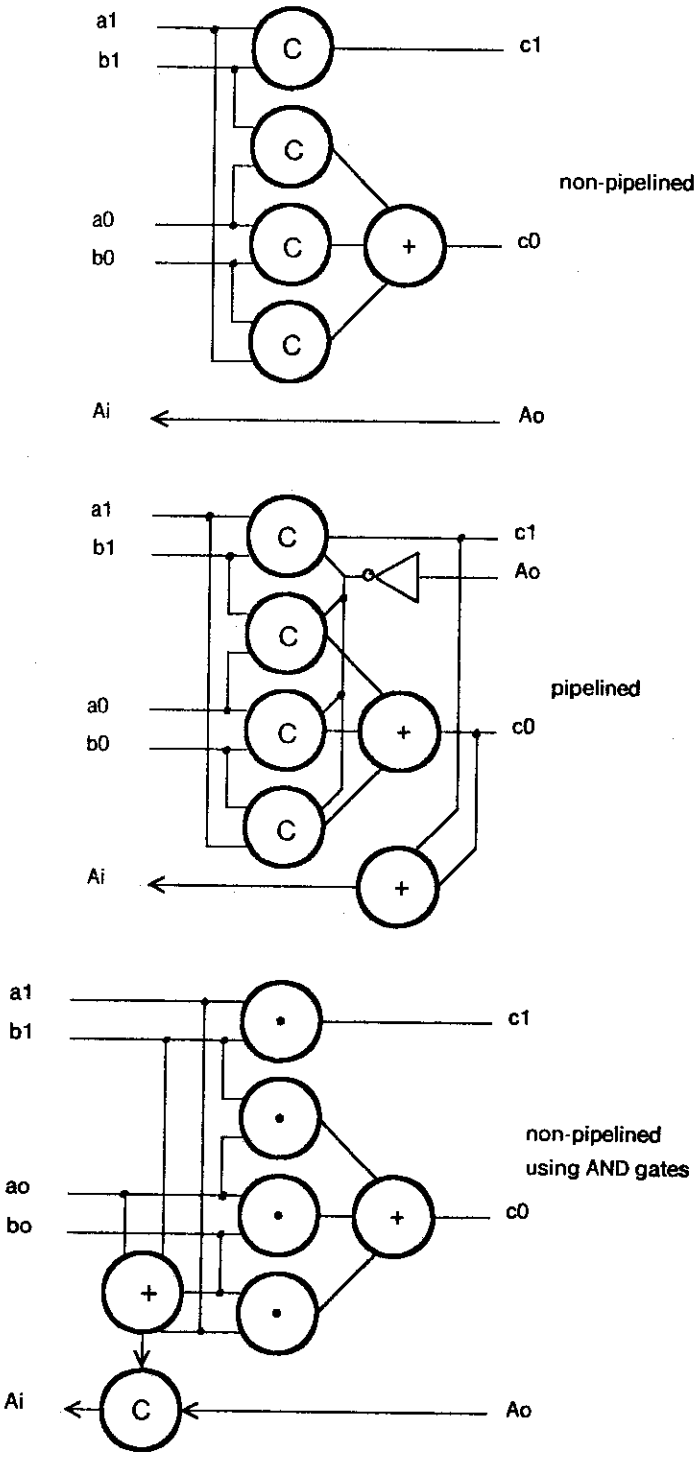
Also, in a sum of product realization, every variable x_i and its complement x'_i are used 2^{n-1} times each. Thus, this type of implementation can be quite expensive.

If combinational logic functions are implemented using this method, we can see that the weak-conditions mentioned earlier are indeed satisfied in the active phase, when input and output signals change from spacer to data state. However, they are not in the reset phase, when the input and output signals return to the spacer state. In order to enforce the weak-conditions in this phase, C-elements are used in place of AND gates. For example, a dual rail AND module shown at the top of Figure 11 has no delay hazards and satisfies the weak-conditions.

Since the C-elements are storage devices, the combinational logic modules can be pipelined in a similar fashion to those designed by Muller [14]. Figure 11 also shows a pipelined version of the AND module, which is almost the same as before with a little additional hardware.

The most simple NMOS circuit for the C-element contains eight transistors, therefore the amount of circuitry required for combinational logic implementation can be excessively large. It is more economical sometimes to implement the combinational logic modules using AND gates instead of C-elements when non-pipelined modules are used. Such an implementation for the AND module is shown at the bottom of Figure 11, in which an OR gate is used to detect the input spacer. In this implementation, the weak-conditions are violated in the reset phase, however, the module is still self-timed and can be connected to other modules to form a correct self-timed system.

Figure 11. Implementations of Self-timed AND Module

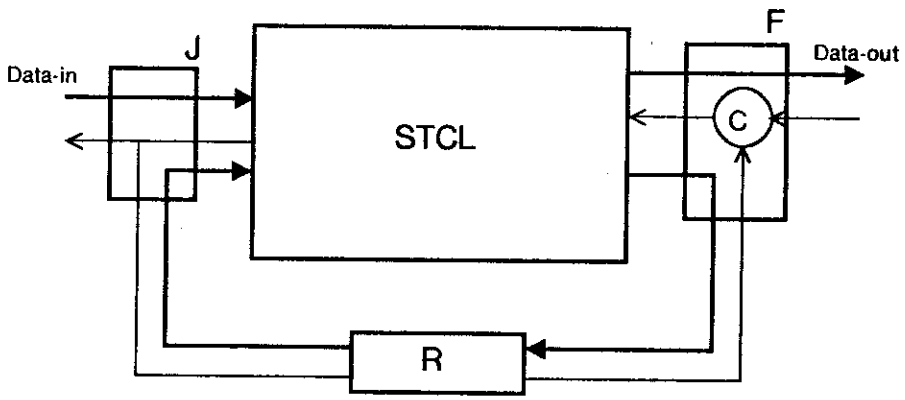


3.3.3 Self-timed State Machine

The following implementation of state machine has been proposed in [19], and is shown in Figure 12. It contains a core of self-timed combinational logic, a feedback register, a fork and a join module. The feedback register contains three register stages, whose implementation is described in [11]. The initial state of the machine is programmed into the middle register stage, sandwiched between two spacer-ed register stages. Such an arrangement is necessary so that once reset, the combinational logic will be in the spacer state.

3.3.4 Other modules

There are a number of nondeterminate and non well-formed modules which are very important for the construction of self-timed systems in general, and of their control structures in particular. The merge module, comprising of an arbiter circuit internally, is such a nondeterminate module; the switch and multiplexor modules are determinate but non well-formed ones. Their implementations can be found in [11]. The question of which modules should be included to form a universal set of modules has been addressed in [9]. For most of the systems we are concerned with, all modules described above together with the source and sink modules [11] form such a set.



J - Join module , F - Fork module

R - Register module , STCL - Self-timed Combinational Logic

Figure 12. Implementation of a Self-timed State Machine

4. Design of the Two by Two Router

In this section, we describe a top down design methodology and its application to the design of the router. This methodology was developed and presented in [11]. In summary, it consists of three main steps :

1. A high level description using a hardware description language is used to specify the behavior and structure of the system. This description can be refined gradually as the design proceeds to produce a more detailed and accurate specification. A description in PADL [10] for the 2x2 router is included in the Appendix.

2. The next step is to translate this description into a data flow graph representation, which can then be directly mapped into the hardware self-timed modules as described earlier.

3. In this step, the final hardware representation of the system is obtained. In this form, the feedback loops and the nodes in the arcs are expanded into their hardware equivalences, being sets of three register stages and the fork and join modules, respectively. One interesting thing to note is that since all combinational logic modules have states (due to the C-elements), the system is initialized by resetting all feedback registers and source modules to spacer state so that combinational logic modules also reset to spacer state.

4.1 Description of the Router Chip

We now describe the physical layout and floor plan of an experimental version of the 2x2 router chip with two-bit wide data path.

The 2x2 router is implemented in NMOS, using Mead and Conway's design rules. Self-timed modules, the basic undecomposable building blocks of self-timed systems, were first designed, laid out and electrically simulated by the circuit simulation program SPICE and then saved in a cell library. The chip design was done by placing these macro-cells and making interconnects between them.

The layout system on which the artwork was generated consists of the symbolic design language DPL [2] and the graphic editing system Daedalus on the LISP machine. Macro-cells were designed and entered graphically through Daedalus. DPL codes were written to place modules and make interconnects. The amount of work had been substantially reduced as everything could be described symbolically and relatively to each other. The final layout had been checked for design-rule violations and simulated, using a switch level simulator.

4.2 Structure and Floor Plan of the Experimental Chip

Structurally, the 2x2 router is composed of four subsystems of two types : Control Modules (CM) and Output Modules (OM), connected as shown in Figure 13. The control section of CM is a simple state machine which takes the address bit (bit 0) of the input packet, decodes it and sends out appropriate control signals to the input switch and request signals to two OMs. Its data path is simply a switch modules which routes the two-bit dual-rail inputs to either the upper or the lower OM. The control section of OM consists of a merge module to arbitrate two asynchronous requests from CM's, and a state machine to control the output multiplexor. Its data path is a multiplexor module, forwarding either the upper of lower packets to the output port.

Figure 14 shows the floor plan and other features of the chip. The one-to-two converters (denoted as 1->2) convert single-rail signals into dual-rail signals; likewise, the two-to-one converters convert dual-rail signals back to normal ones. The merge modules (M) of om's are shown explicitly, from which test points are connected. The small squares denote the test buffers, allowing external access to control signals of the modules. Their structures will be presented later. In this chip, input queues are not included as should be, so that access to the data path can be readily made.

The test chip has 40 pins, 20 are reserved for active signals, power and ground. The other 20 are test pins, providing access to the control signals. Four test pins monitor the outputs of the arbiters in the merge modules, they allow thorough testing and evaluation of the arbiters' characteristics. The active signals are :

**lbin-u, lbin-d, lbout-u, lbout-d* : 'last byte' signals of the input and output ports.

**din-u, din-d, dout-u, dout-d* : data signals. For this chip, each port has only one data bit,

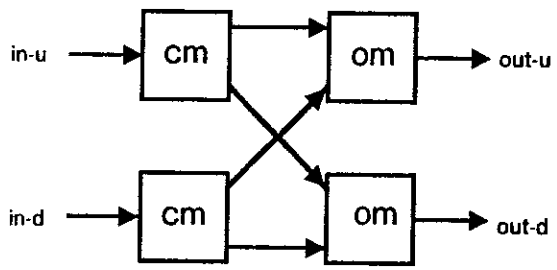


Figure 13. Structure of the 2x2 Router

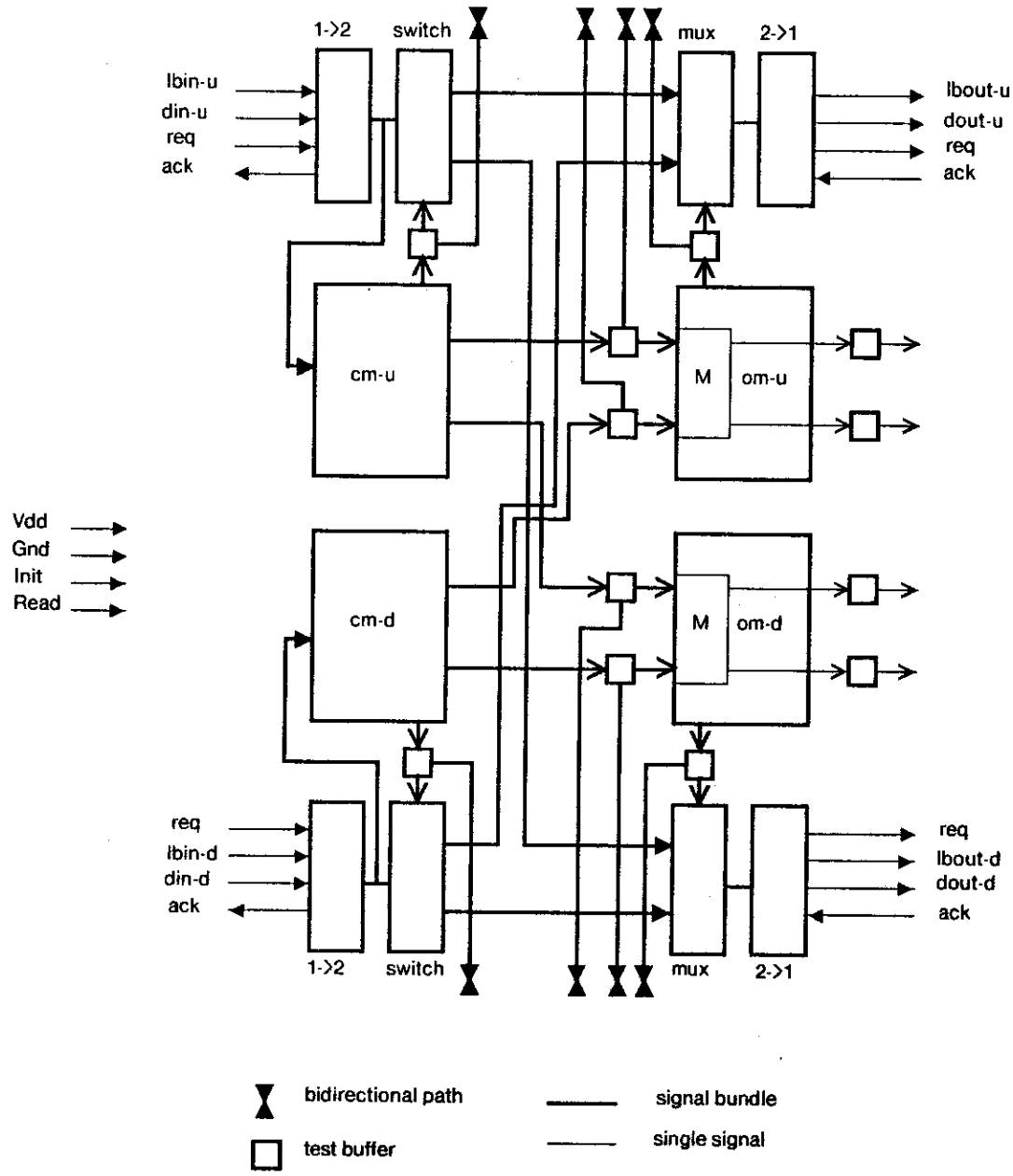


Figure 14. Floor Plan and Test Buffers of the 2x2 Router

---

however, the data path width can be expanded without altering the design of the control parts.

**Req,Ack* : request and acknowledge signals to implement the reset signaling protocol for the ports.

**Init* : the initialize signal. When asserted, all registers are set to their initial states, all source modules to their initial values.

**Read* : this signal is used for testing. When *read* = 1, the test buffers output the control signals to bidirectional i/o pads. When *read* = 0, external signals can be connected to the control lines of the modules.

4.3 Struture of CM and OM

The layouts of CM and OM are described in the following short paragraphs. Modules are shown as blocks, but their locations, sizes and interconnections reflect the actual layout. Figure 15 shows the block layout for CM, whose control section consists of a state machine, a bit delay, an F-gate (constructed from a switch module and a sink module), and another switch module. The state machine is constructed from three NAND modules and a feedback register. This register consists of three self-timed register stages, the middle one is initialized to zero-data state, the other two spacer state. The data path of CM is simply a two-bit wide switch module.

Figure 16 is the block layout for OM. The control section is slightly more complicated. The merge module accepts two request tokens (which are actually direction bits) from two OM's, passes one and blocks the other until the first one disappears. The winning token is sent to the lower mux module and T-gate, and then cycled through the feedback path *feedback-a*. The other two feedback paths and the upper mux module monitor the *lb* token and reset the direction bit when the last byte of a packet has gone through the router. The data path of OM is simply a two-bit wide mux module.

4.4 The Test Buffers

Two types of buffers are used to bring important signals to test pins. They are the unidirectional and bidirectional buffers. Their symbols and implementations are shown in Figures 17 a, 17 b and 17 c. The unidirectional buffer is a non-inverting one, it provides a buffering stage between the internal signal and the i/o pad and allows signals to be observed externally. The bidirectional buffer can be configured in read or write mode, as shown in Figure 17 b. Its implementation (Figure 17 c) requires a level-restoring buffer, a MOS switch, and a bidirectional i/o pad with tristate buffers controlled by the *read* signal.

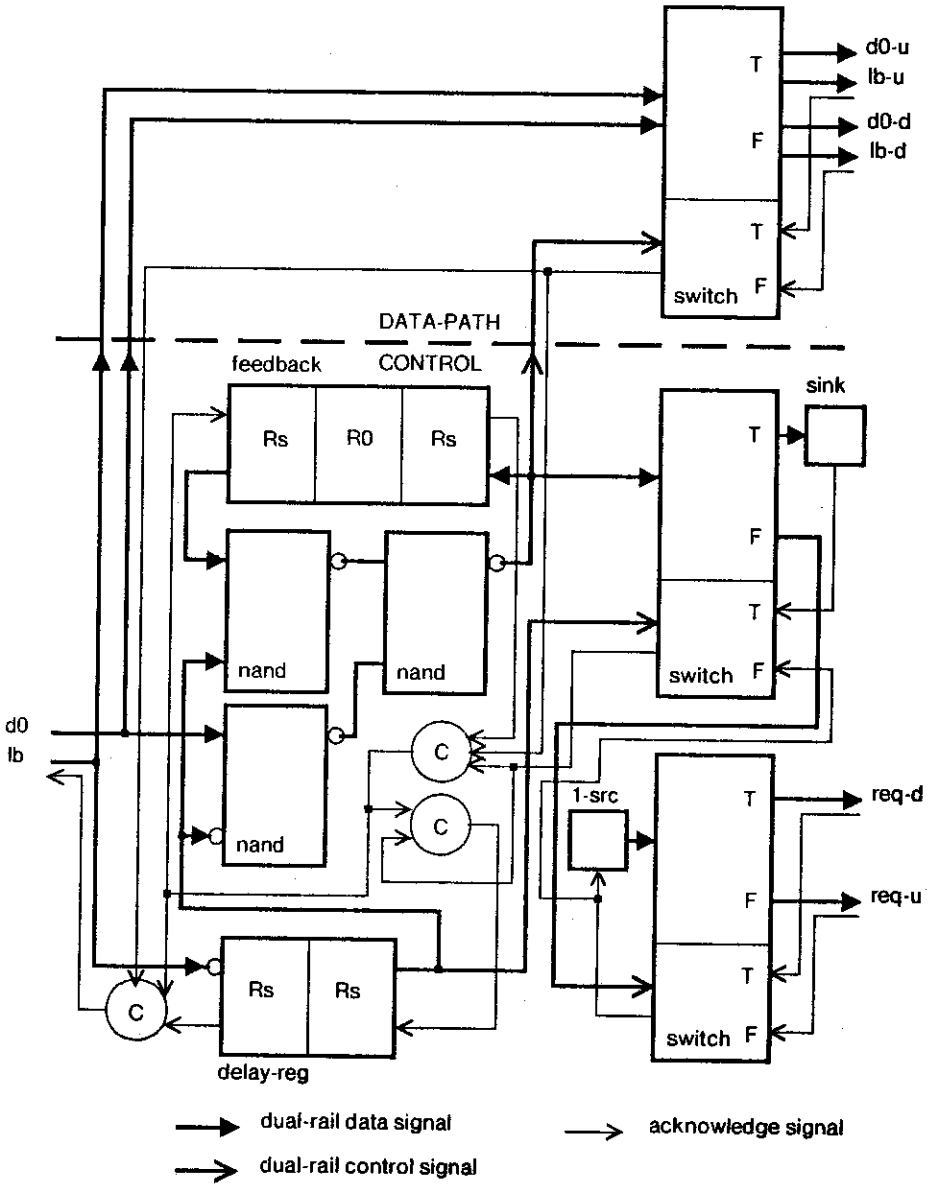


Figure 15. Block Layout of Control Module (CM)

4.5 Power and Signal Routing

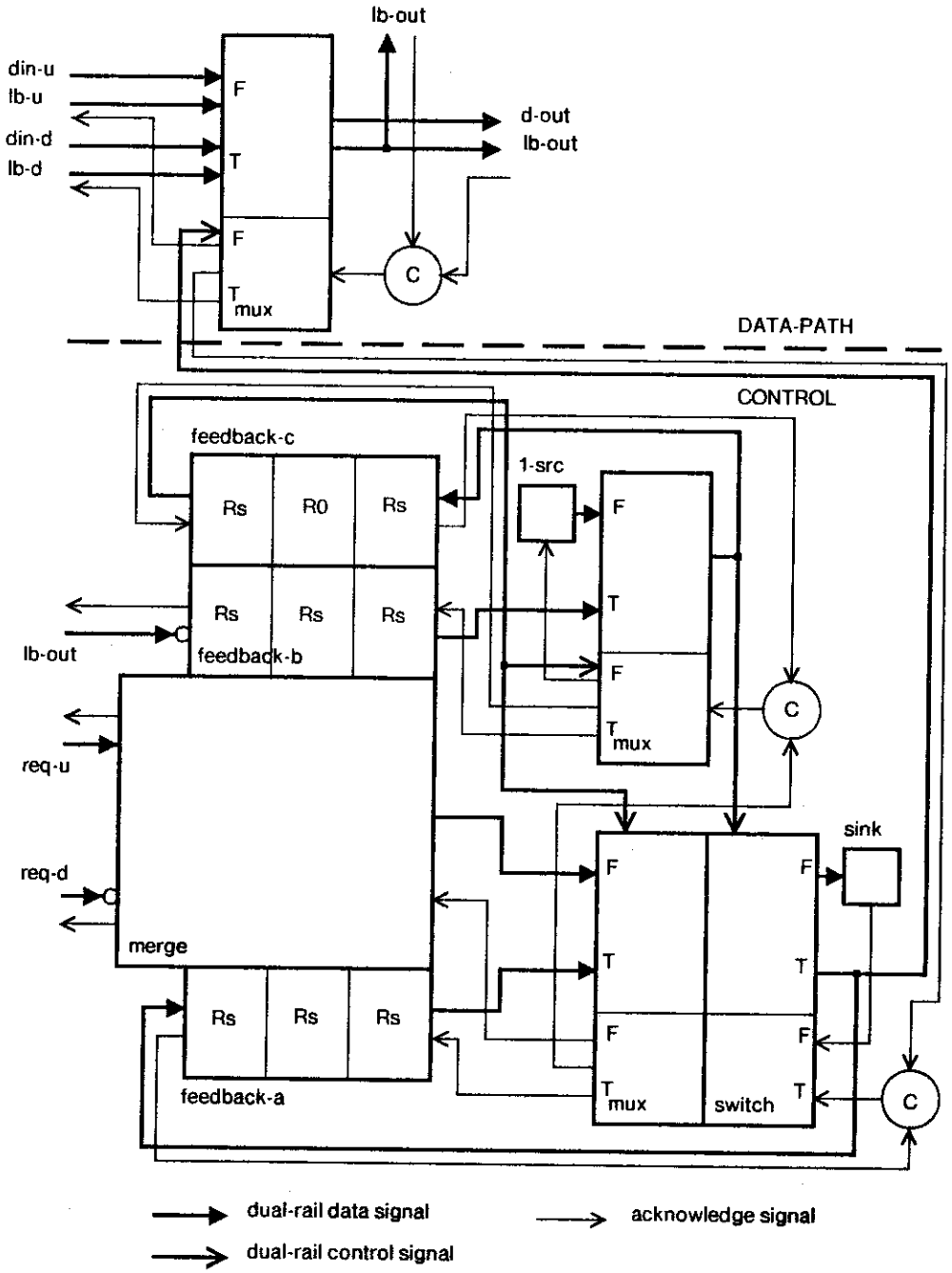


Figure 16. Block Layout of Output Module (OM)

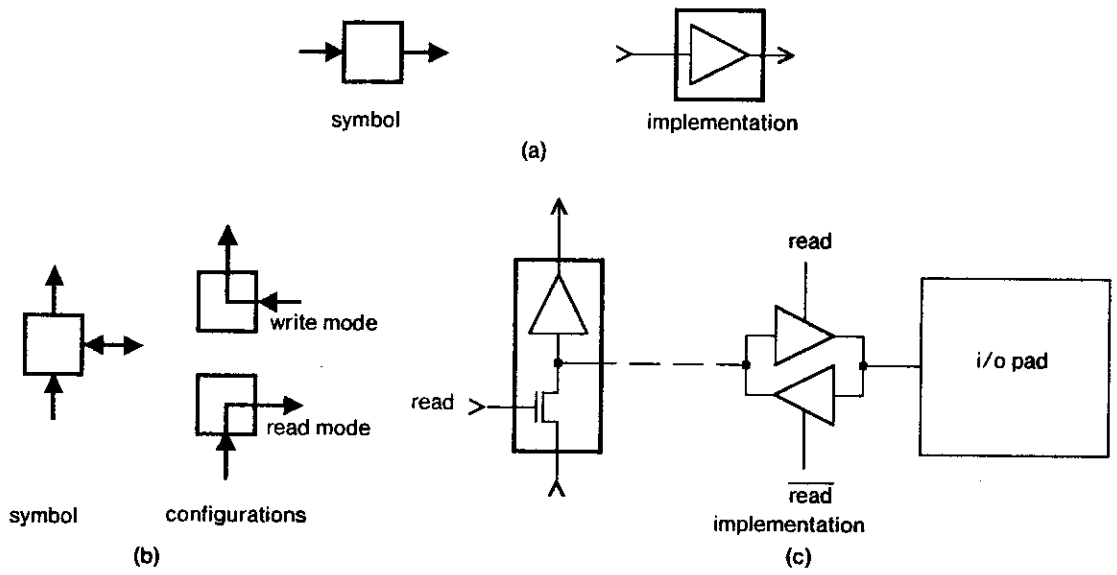


Figure 17. Symbols and Implementations of the Test Buffers

Two requirements of the router design make the routing problem important : first, the use of dual-rail coded signals increase the number of wires needed by at least a factor of two over conventional design; secondly, interconnection of subsystems of the router requires some attention in layout so that both straight and cross-over connections between modules can be made using a minimum amount of area.

Since the first problem is inherent in the design methodology, one acceptable solution is to use single rail signals with explicit *Ready/Acknowledge* or other methods of encoding. Certainly, in using single rail signals, one of the basic assumptions of the methodology has to be relaxed : one now has to be able to guarantee that he can account for all gate and wire delays within small regions of the chip. This is usually possible in the real world as a chip may contain many subsystems, each small enough to conform with the above requirement. It is noted here that the use of dual-rail signals results in penalty in area, but not in terms of the time required to perform the routing, as dual-rail signals always travel in pairs.

Routing of power buses for the chip is also aimed at facilitating signal routing between subsystems. Usually, power and ground buses are made as inter-digitated forks. In the router chip, the power buses are in a formation which can be termed "two-level forks", as shown in Figure 18. By doing this, an empty region can be reserved for channel routing, where signals can be routed on both metal and polysilicon layers. Another advantage of such power busing scheme is it allows some hierarchy in power signal routing, corresponding to the same levels of hierarchy of the subsystems. The power buses provided for each subsystem need only produce enough current as required. Note that this is not necessarily true for regular structures such as PLA's, where power buses have to be large enough to sustain the maximum current requirement.

Routing between subsystems composing the router is another nontrivial issue, as it is necessary to make both the straight and cross-over connections. The cross-over connection is made as in Figure 19, since the vertical distances between subsystems are much larger than the horizontal spacings, vertical interconnects are run in metal, horizontal ones in polysilicon. This helps minimizing delay between port signal and improves performance, even though the design should be insensitive to delays in wires connecting modules. For this kind of channel routing, if there are N signals at each port, the width of the routing channel is 3N.

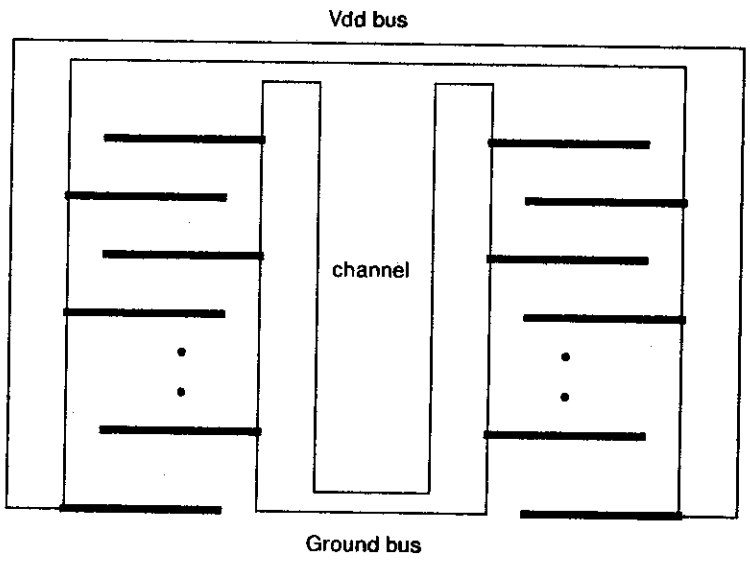


Figure 18. Power Bus Routing for the Router

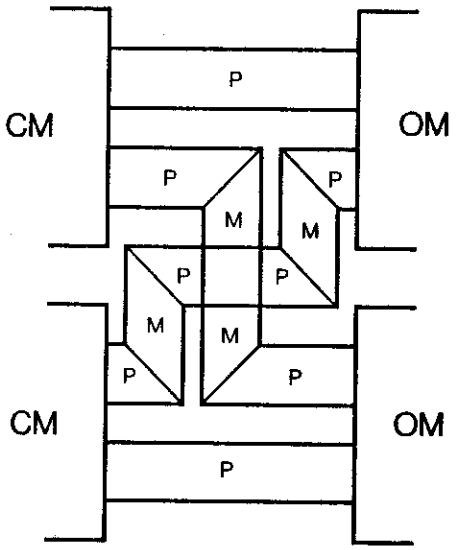


Figure 19. Channel Routing of the Router

5. Design of the Test Circuit for the Two by Two Router

A test circuit for the router is being designed and constructed. The test circuit has a number of extra functions designed solely for this experimental chip, and not required in the final version. The test circuit itself is synchronous and has interfaces to a PDP11. This allows automatic control and testing of the router from the PDP11. Since the router is a self-timed chip with Ready/Acknowledge signals at input and output ports, it is necessary to design the control circuit for the synchronous/asynchronous interface.

Two types of test will be administered on the router chip : functional and electrical. In functional testing stage, the function of the router is checked by transmitting data packets to the chip, reading them back and comparing them in the computer. All configurations of the router (cross connection, straight connection, contention) will be tested by setting the address bits of the data packets. With the additional test pins provided, functional testing can be carried out with ease. For electrical testing, the test circuit will be designed to allow the router to operate at its maximum rate, from which delays and throughput rates can be estimated. Also, the electrical characteristics of the arbiters residing in the Merge modules can be determined as test pins allow access to their inputs and outputs. Before showing the design of the test circuit, we present a fault model and discuss the effects of such faults on the router. The test circuit should be able to detect a large number of faults if they occur at all. As it turns out, the characteristics of self-timed modules and the use of dual rail code tremendously aid the testing process, and we claim that a simple test set for multiple stuck-at faults can have one hundred percent test coverage for the router.

5.1 A Fault Model and Its Effects

We assume the most simple type of faults for digital systems : stuck-at faults. In reality, the more common type of faults in integrated circuits are bridging faults, corresponding to signal wires shorted to each other. Stuck-at faults are special cases where signal wires are shorted to either ground or the power supply. In general, bridging faults are difficult to detect as the effects of such faults vary. A bridging fault might manifest as a stuck-at fault or an intermittent fault in the case when the input and output of a combinational logic circuit are shorted together and cause an analog voltage level. Thus, it is desirable to be able to deal with stuck-at faults alone, as this would greatly simplify the task of detecting faults. By using some layout techniques as discussed in [4], we can reduce most bridging

faults to stuck-at and analyze their effects on the system.

Consider now what happens if a single stuck-at fault occurs in a self-timed system. First, it is claimed that for self-timed modules belonging to the set described earlier, any stuck-at fault inside them can be modeled as stuck-at fault at their input or output. Part of the reason is that the self-timed modules described are not state-intensive, and the states of C-elements are in general accessible from either the input or output of the modules. This can be verified by simply checking all the modules exhaustively. Also, due to the use of the dual rail code, we need only consider self-timed systems with stuck-at faults on data and acknowledge signals. We will deliberate on several cases, correspond to stuck-at-1, stuck-at-0 faults and to whether the faults are on data or acknowledge signals.

5.1.1 Effects of Stuck-at Faults on Acknowledge Signals

If an Acknowledge signal is either stuck at 1 or stuck at 0, the communication between modules using the reset signaling protocol can not be executed to a completion, i.e., a signaling cycle may not return to its reset phase. This breakdown in communication at a local link between two modules will eventually propagate to other parts and to the input and output of the system. Thus, the whole system will hang up and no further operation is possible. Note that in the case of stuck-at faults on acknowledge signals, it is not important whether a stuck-at-0 or stuck-at-1 fault occurs, the effect will be the same.

5.1.2 Effects of Stuck-at Faults on Data Signals

Because of the way data are coded, stuck-at-0 and stuck-at-1 faults have slightly different effects on the system. In dual-rail code, the 00 state corresponds to the spacer, while the 11 state is an illegal state which can cause illegal output. If an input signal is stuck-at-0, it can cause either an incompletely or a completely defined input, depending whether that signal wire has to go to 1 or not. In the first case, the output of the module will be a spacer due to the constraint of the weak-conditions and the method of synthesizing combinational logic (any incompletely defined input is mapped to an output spacer). In the latter, a correct output is produced. Thus in order to detect a stuck-at-0, one needs to excite each coded input with both a zero- and a one-data.

For a stuck-at-1 input signal, during the active phase of the signaling cycle, either a correct or an illegal output will be generated. However, in the reset phase, the output will not reset to a spacer because the stuck input will not return to the spacer state. This will hang the communication at the module and subsequently of the whole system. Hence, the detection of stuck-at-1 faults requires only one input excitation, either by a zero- or a one-data.

We can now study the more general case of multiple stuck-at-faults in the system. Beside being a delay insensitive method of data transmission, the dual rail code is a error-detection code. One particularly useful property of this so called auto-synchronous code [1] is that the encoding or decoding is bit-wise independent, i.e., each coded signal is isolated and independent of the other. This property allows us to deduce that the effect due to multiple stuck-at faults in a self-time system is the collection of effects due to each individual fault. This implies that each stuck-at-fault can be detected independently of others. For example, a direct consequence of this property is that for a module with n-bit coded input, a complete test set contains two n-bit vectors (e.g.,an all one-data and an all zero-data vector) for stuck-at-0 faults and one vector (any valid input code) for the stuck-at-1 case.

5.2 A Test Strategy

The following statement summarizes the above discussion about fault modeling and detection in self-timed systems :

If a self-timed system is constructed from self-timed modules with characteristics described in the previous section, and if stuck-at faults are assumed, then

> Any stuck-at fault on acknowledge signals is detectable by exciting the module having the stuck-at faults once by any valid coded data vector.
>
> Any stuck-at-1 fault on data signals is detectable by exciting that data port once with any valid coded data vector.
>
> Any stuck-at-0 fault on data signals is detectable by exciting that data port with two valid coded data vectors, one being the complement of the other.

For these cases, faults are detected by inducing the system into hung-up states in which no further communication and processing is possible, as the presence of stuck-at faults will finally freeze the communication of the system.

In particular, for the router, we can use the following test sequence. The minimum length of a packet is two bytes, with the first being the address, the second the data byte. In order to detect all stuck-at-faults, an arbitrary coded vector X and its complement X' are needed. From each input port, we need to check the straight and the cross-over paths to the output ports. This requires two packets, one containing two bytes XX', the other X'X, where X is any valid coded data byte. Since there are two input ports, the total number of test bytes is eight. However, parallel checking can be carried out at two input ports using, for example, the following test sequences :

	First packet	Second packet
Input port 1	XX'	X'X
Input port 2	X'X	XX'

Such a simple test will cover all multiple stuck-at faults in the router.

5.3 Details of the Test Circuit

The test circuit is composed of the following components : two input ports, two output ports and the bus interface circuit, as in Figure 20. The input and output ports are driven by the same clock.

Each input port contains a set of shift registers. Data are loaded from the Unibus into these registers, then they are shifted into the input of the router when a Go control signal is sent from the computer. A simple state machine is used to provide the handshake signals and also additional functions such as Go/Stop, Hold/Cyclic. The state machine receives control signals from the bus and the asynchronous acknowledge signal from the router, it sends control signals to the shift registers. Go/Stop will start or stop the shift registers, Hold/Cyclic will allow the input ports to either send a fixed number of packets or keep cycling the packets and send them to the router continuously.

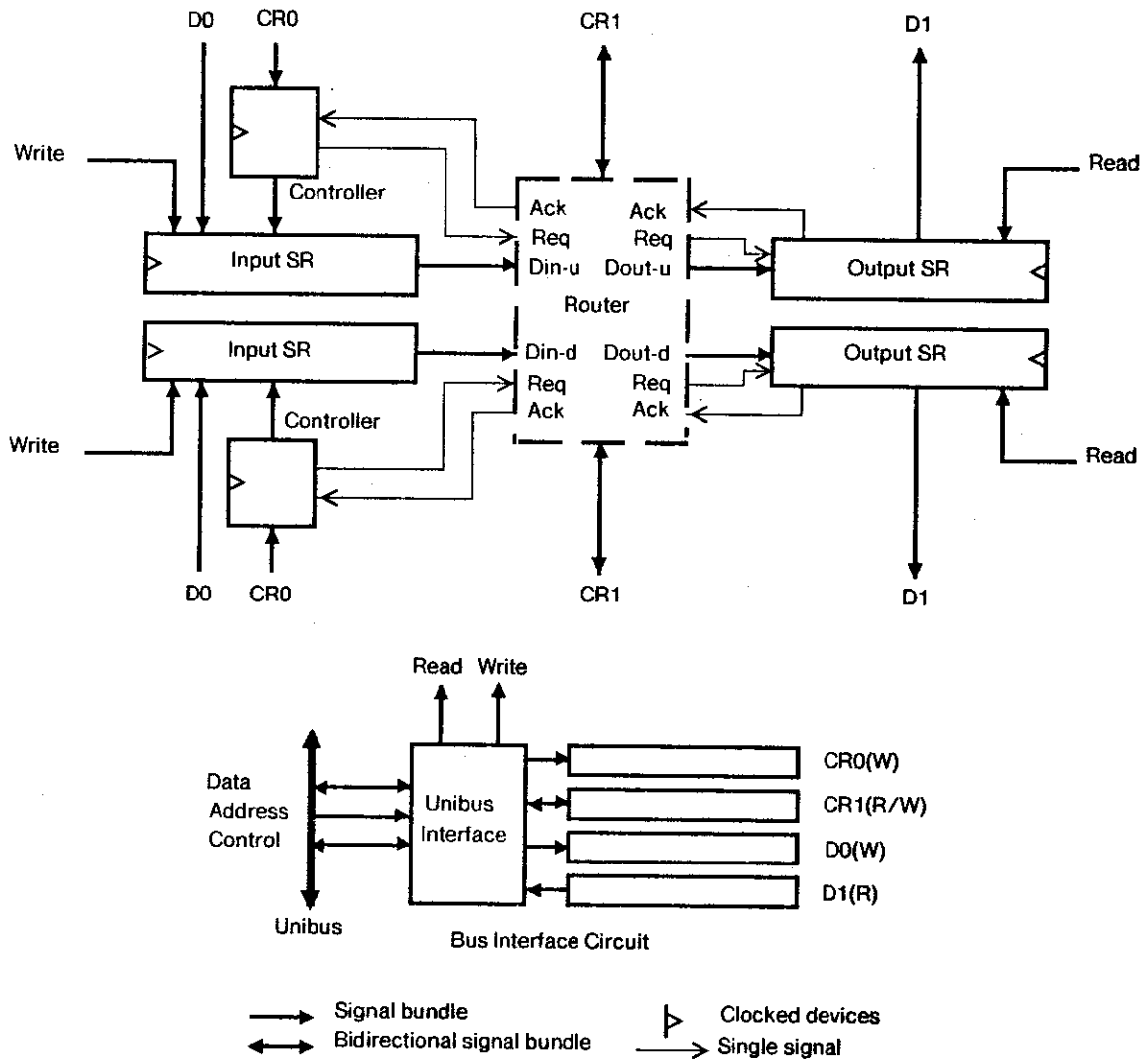


Figure 20. Block diagram of the Test Circuit

The output ports are simply shift registers with one extra flip-flop to generate the acknowledge signal. Data are loaded into them by the request signal coming out of the router. When the shift registers are full, data can be read back to the computer through the bus. Since one does not know exactly when all data are shifted out to the registers, a time-out mechanism can be used. The computer waits a fixed amount of time – long enough for the output shift registers to be filled by the test chip in normal operation, and starts reading their contents. If stuck-at faults exist and hang the system under test, the output data will be faulty and different from the input data sent to the chip.

The bus interface circuit consists of the address selector and the bus control signal decoders. Four bus addresses are dedicated for the test circuit. There are two control registers and two data registers. The first control register (CR0) is a write-only register, it has 5 bits for selecting a pair of input and output shift registers (out of sixteen) for writing or reading data, the rest is used to control the function of the input ports (Go/Stop, Hold/Cyclic). The second control register (CR1) is a read/write register connected to the test pins of the router, thus allowing access to the internal signals. In the final version of the test circuit, CR1 will not be used at all. Two data registers D0 and D1 are used for the input ports and output ports. D0 is write-only, allowing data to be loaded into the input shift register whose address is selected by CR0. D1 is read-only, allowing data to be read by the computer from the output shift register selected by the same address in CR0. Note that D0 and D1 are actually sets of (shift) registers, however, they are multiplexed to look like single ones to the bus.

5.4 The Asynchronous Interface Circuits

Few words should be said about the reliability of the asynchronous interface circuits between the input/output ports and the router. In the above design, asynchronous handshake signals from the router are synchronized using clocked flip-flops. There is always a chance that this scheme will fail due to the behavior of the latch in metastable state. However, the probability of failure is so small that it can be neglected. The reason is that the NMOS router is expected to be slow compared to the test circuit built from high speed components such as Schottky TTL (or even low power Schottky TTL).

6. Evaluation

In this section, we evaluate several aspects of the self-timed design methodology described above. By comparing the self-timed implementation of systems to other standard approaches, specifically, the synchronous system design methodology, we can determine the advantages and drawbacks of this methodology. One of the most important performance-related measures of digital integrated circuits are area, speed and power consumption. Other indicators are ease of design and modification, optimization, and verification or testing.

6.1 Area, Speed and Power Consumption

Area consumption is perhaps one negative aspect of self-timed circuit design. Based solely on the fact that dual-rail code is used, the area required for circuit implementation is at least twice as much as that of a similar synchronous design. Also, the implementation of the self-timed communication scheme results in expensive realization of combinational logic and other modules.

In terms of speed, it is strongly believed that self-timed circuits will offer large improvement over synchronous design. There are two main reasons : first, since only local communication is allowed, individual modules dictate the speed of operation, and therefore it reflects the average speed of the modules instead of the slowest ones. Secondly, as mentioned earlier, delays due to long distant communication will dominate the circuit delays as (MOS) devices are made smaller, however, such type of communication does not exist in self-timed systems. It has been and will remain true that the motivation for going to asynchronous design is the greed for speed.

A NMOS implementation of a self-timed system would consume at least twice the amount of power required for a corresponding synchronous design, as the power consumption is approximately proportional to the amount of circuitry used. Another reason is the fact that self-timed circuits are inherently static, therefore no dynamic techniques available in MOS technologies can be exploited. However, this problem is not as serious if a low power technology such as CMOS is used.

6.2 Ease of Design and Modification

Two important characteristics of the above design methodology are : it is modular and it is a top down approach that reflects the hierarchical structure of the system. These imply that design of systems using the self-timed approach is easier because it adheres to the way the designer thinks and proceeds from the top level all the way through, up to a low level of hardware representation. On the contrary, a resulting synchronous system may bear little resemblance to the conceptual picture the designer had in mind. Also, since the system is modular, modification is easier and simpler as changes are local to certain modules and do not affect the system globally.

6.3 Optimization

Low level optimization is very limited for self-timed systems in general. The bare fact that the system is constructed from self-timed modules places severe restrictions on what kind of optimization is allowed. As mentioned earlier, since a clock is not available, no dynamic techniques can be afforded, and all circuits have to be static. This is a big drawback if MOS technologies are used, because the basic logic forms do not include the pass transistor logic. Due to the same reason, precharging, bootstrapping are not realizable. However, optimization at higher level is easier and more straightforward for self-timed systems, as the modularity and uniform characteristics of the system modules give the designer more flexibility in constructing and altering the network of modules intended to implement the system.

6.4 Verification and Testing

As discussed in [5], correct operation of a self-timed modules implies correctness of function, and of their electrical and temporal specifications. Once the modules are designed and guaranteed to have no local logical or timing errors, no unanticipated timing problems will occur when systems are put together from modules. Because of that, correct operation of self-timed systems are verifiable from a high level simulation; one is freed from worrying about the timing problem of the system. In a synchronous system, precharged buses, shared buses are some of numerous sources of timing problems which can not be verified by simple simulations.

In terms of testing, as presented earlier, one can obtain order-of-magnitude gain in testing of and test generation for self-timed systems. Two important consequences of the self-timed discipline in terms of testing are the following. First, stuck-at faults always induce hang-ups in a system. Secondly, effects of multiple stuck-at faults can be considered as the collection of effects contributed by individual faults. These permit the detection of multiple stuck-at faults using short and simple test sets.

7. Conclusion

Despite of many advantages of self-timed design, one single most critical drawback of the approach, experienced from this project is the large amount of area consumed by the final integrated circuit. This had led us to reevaluate the basic assumptions on gate and wire delays, and found that perhaps a less stringent assumption could allow more practical and area-efficient implementation. After all, logic gates are elements with finite and bounded response times (otherwise, synchronous systems could not be realized at all); the only case where circuit with unbounded delay exists is the arbiter circuit in metastable state. If the potential errors caused by the metastable state are kept below some acceptable bound, then more efficient self-timed systems can be designed under the bounded gate and wire delay assumption. A preliminary study has shown that an array structure approach is particularly attractive, as the uniform nature of the array allows one to account for the delays accurately at a high level of system representation. Certain circuit topology such as CMOS Domino logic can be readily incorporated into array structures to form self-timed logic arrays.

8. Acknowledgements

I would like to thank Prof. Jack Dennis for his continued supports and encouragements. Dr. Clement Leung, currently at Patil Systems, is gratefully acknowledged for his insightful and helpful discussions. Willie Lim is thanked for his participation in discussions about fault modeling and testing of self-timed systems. Bill Ackerman and Andy Boughton read the drafts and gave valuable comments. Daniel Weise of the AI Lab, Steve McCormick and Robert Armstrong of the VLSI Lab gave considerable help with the layout of the router chip.

9. References

1. Armstrong, D B et al, Design of Asynchronous Circuits Assuming Unbounded Gate Delays, *IEEE-TC* Vol C-13, No 12, Dec 1969, pp 1110-1120.

2. Batali, J. and Hartheimer, A., The Design Procedure Language Manual, MIT Department of EECS, VLSI Memo 80-31, September 1980.

3. Chaney, T J and Molnar, C E Anomalous Behavior of Synchronizer and Arbiter Circuits, *IEEE-TC*, Vol C-22, April 1973.

4. Chu, T, A Design Strategy for Testable Self-Timed Systems, MIT-LCS CSG Memo 216, April 1982.

5. Chu, T Circuit Analysis of Self-Timed Elements for NMOS VLSI Systems, MIT-LCS, TM-220, May 1982.

6. Dennis, J B, Modular, Asynchronous Control Structures for a High Performance Processor, Project MAC, MIT.

7. Dennis, J B Packet Communication Architecture, *Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing*, IEEE, New York 1975, pp 224-229.

8. Dennis, J B and Patil, S S ,Speed Independent Asynchronous Circuits, CSG Memo No. 54, Project MAC, MIT, Jan 1971.

9. Keller, R M, Toward a Theory of Universal Speed Independent Modules, *IEEE-TC* Vol C-23, No 1, pp 21-32, Jan. 1974.

10. Lim, W and Leung, C PADL – A Packet Architecture Description Language, MIT-LCS, CSG Memo 221, Sept. 1982.

11. Leung, C K C, On a Design Methodology for Packet Communication Architectures Based on a Hardware Design Language, in *Computer Hardware Description Languages and their Applications*, edited by Breuer and Hartenstein, North-Holland Publishing Co, Amsterdam, Holland 1981, pp 171-184.

12. McNaughton, R, Finite Automata and Badly Timed Elements, U. of Pennsylvania, Moore School of Electrical Engineering, Philadelphia, Pa.

13. Misunas, D, Petri Nets and Speed Independent Design, *Communication of the ACM*, Aug 1977, Vol 16, Number 8, pp 474-481.

14. Muller, D E, Asynchronous Logics and Application to Information Processing, *Switching Theory in Space Technology*, Stanford U. Press, Stanford, Ca.

15. Patil, S S, An Asynchronous Logic Array, Technical Memo 62, Project MAC, MIT, May 1975.

16. Patil, S S, Cellular Arrays for Asynchronous Control, CGS Memo 122, Project MAC, MIT, April 1975.

17. Seitz, C L, Self-timed VLSI Systems, *Proceedings of the CalTech Conference on VLSI*, pp 345-355, Jan. 1979.

18. Seitz, C L, System Timing, Chapter 7 of Mead and Conway's *Introduction to VLSI Systems*, Addison Wesley, 1980.

19. Singh, N P, A Design Methodology for Self-Timed Systems, MIT-LCS TR-258, Feb 1981.

20. Sutherland, I E et al, The TRIMOSBUS, *Proceedings of the CalTech Conference on VLSI*, Jan. 1979, pp 395-427.

21. Unger, S H, Asynchronous Sequential Switching Circuits with Unrestricted Input Changes, *IEEE-TC* Vol C-20, No. 12, December 1971.

Appendix

The following is a detailed description of the 2x2 router in PADL. The structure part describes the interconnection of major components, consisting of the control modules (CM) and output modules (OM). The behavior of CM and OM are described in the behavior part of the program.

```
type Router = module(inlet IN_U[0:8], IN_D[0:8] : bitstr;
                     outlet OUT_U[0:8], OUT_D[0:8] : bitstr)

        submodule cm_u, cm_d : cm;
        submodule om_u, om_d : om;

% Structure of the 2x2 Router

structure
            IN_U -> cm_u.IN ;
            IN_D -> cm_d.IN ;
%
            cm_u.OUT_U -> om_u.IN_U ;
            cm_u.OUT_D -> om_d.IN_U ;
            cm_u.REQ_U -> om_u.REQ_U ;
            cm_u.REQ_D -> om_d.REQ_U ;
%
            cm_d.OUT_U -> om_u.IN_D ;
            cm_d.OUT_D -> om_d.IN_D ;
            cm_d.REQ_U -> om_u.REQ_D ;
            cm_d.REQ_D -> om_d.REQ_D ;
%
            om_u.OUT -> OUT_U ;
            om_d.OUT -> OUT_D;

endstruct;
```

```
%
% Behavior description of Control Module (CM).
%
type CM = module(inlet IN[0:8]: bitstr;
          outlet OUT_U[0:8], OUT_D[0:8], REQ_U, REQ_D : bitstr)

%
% Variable First indicates the first byte of a packet.
% Variable Dir is the direction bit of the packet.
%

          UP, DOWN : bitstr := '0, '1;
          var   First : bitstr := '1;
                Dir : bitstr := UP;
  cycle
     let pkt = from IN
     in
          ODIR : bitstr := if First
                           then pkt[0]
                           else Dir endif;
          if First then
                          if ODIR = = UP
                          then begin
                                 send UP at REQ_U;
                                 send pkt at OUT_U;
                               end;
                          else begin
                                 send DOWN at REQ_D;
                                 send pkt at OUT_D;
                               end;
                          endif;
%
          else
                          if ODIR = = UP
                          then send pkt at OUT_U
                          else send pkt at OUT_D
                          endif;
          endif;
%
     Dir := ODIR;
     First := pkt[8];
     endlet;
  endcycle;
endmod;
```

```
%
% Structure description of Output Module (OM)
%

type OM = module(inlet IN_U[0:8], IN_D[0:8], REQ_U, REQ_D : bitstr;
                 outlet OUT[0:8] : bitstr)

%
% Variable Odir indicates the direction the output packet comes from.
% Variable First_Not is true when the byte is not the first byte of a packet.
%

        UP, DOWN : bitstr := '0,'1;
        var   Odir : bitstr;
              First_Not : bitstr := '0;

  cycle

    let IDIR : bitstr := if ~First_Not
                         then from_either IN_U, IN_D
                         else Odir
                         endif;
    in
        let   pkt :=     if Idir = = UP
                         then from IN_U
                         else from IN_D
                         endif;
        in    send pkt at OUT;
        endlet;

        Odir := IDIR;
        First_Not := ~pkt[8];

    endlet;
  endcycle;
endmod
```