Design and Evaluation of two Asynchronous Token Ring Adapters

C.Carrión[*]
Dpt. Electrónica y Computadores
Uni. of Cantabria
39005 Santander, Spain
Phone: +34-42-201399
Fax: +34-42-201402
e-mail: carmen@atc.unican.es

A. Yakovlev[†]
Dept. of Computing Science
Uni. of Newcastle upon Tyne
NE1 7RU, England
Phone: +44-191-222-8184
Fax: +44-191-222-8232
e-mail: Alex.Yakovlev@ncl.ac.uk

Abstract

The goal of this paper is to design and measure the performance of two speed-independent asynchronous adapters for a communication architecture with distributed arbitration. The protocol is defined on the principles similar to those of the Token Ring but attempts to exploit the asynchronous design approach in improving the operational scalability of the system. Its two versions considered in this paper reflect the tradeoff between the greater functional capability and low-latency requirements. We show the power of Signal Transition Graphs in specifying, verifying and synthesizing the adapter's logic. Using timing information obtained at the implementation level, we finally estimate the performance of the token-ring architecture with point-to-point interconnection.

keywords: *arbitration, asynchronous systems, delay-insensitive communication, signal transition graph, speed-independent circuits, token ring adapter*

1 Introduction

The use of asynchronous logic seems unavoidable in applications involving dynamic computational processes that can be activated by non-deterministic input signals. A clear example is communication processes where the transmission of information can begin at any instant. The use of synchronous logic in the design of the Medium Access Control (MAC) layer interface, according to ISO classification of multi-layered Local Area Networks [8], implies the existence of a local clock in each adapter and a synchronization process in the interface that enables the exchange of information correctly. Such synchronization, if based on the global clock, creates obvious problems with scalability and robustness of the system, letting along the timing overhead concerned with efficient clock distribution. There is a clear tendency amongst the designers of new systems to build scalable coherent communication media even at fairly low levels of systems architecture, such as e.g. memory interface (RamLink) [5]. There have recently been examples of exploiting asynchronous design methods to tackle this problem. A scalable asynchronous interchip bus, based on a three-wire protocol, has been proposed in [15].

Compared to buses, ring-based architectures offer additional advantages to the scalability factor, though for the price of latency [5]. An example of an asynchronous ring interface of the MAC level has been presented in [21]. Its protocol was however not quite close to the commonly used Token Ring one. The arbitration method was rather inefficient and separate from the data transmission, resulting in higher latency and lower rate for short messages. The speed-independent logic implementation was quite complex because its control part was obtained by means of the direct translation of its Petri net model. That was quite understandable as at the time when that design was produced there were no adequate automated logic synthesis tools for Petri nets.

[*]Supported by CICYT, Spain, grant TIC95-0378.
[†]Supported by EPSRC, UK, grant GR/K70175.

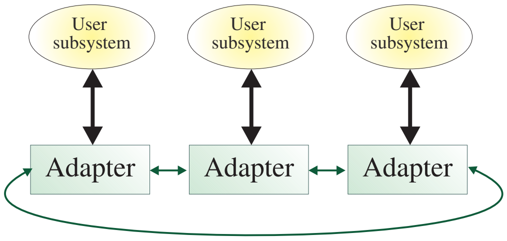


Figure 1: Ring structure

The goal of this paper is to design a speed-independent asynchronous adapter for a ring channel with token-based arbitration, avoiding the above-mentioned problems in [21]. The new designs exploit the existing automatic tools for verification and logic synthesis from Signal Transition Graphs (STGs) [10] [6]. The tools we have used are SIS [10], ASSASSIN [11], Versify [14], Petrify [3]. The use of tools allows more efficient design process and exploration of several options different in functional capabilities and performance. Therefore the additional goal of this work has been to investigate two adapter designs with different addressing mechanisms and analyse their performance, using the data obtained at the implementation level. An important question answered by this work is what the gain in latency would be if the channel protocol did not use address manipulation (an "always-write" method). The answer is that such a gain is not very much – only about 25-30%.

The structure of this paper, is as follows. Section 2 describes the pipeline ring protocol, in which messages travel through the net arriving at their destination without either deadlock or starvation problem. Section 3 presents the Signal Transition Graph as the formal model of asynchronous circuits that will be used in the following sections, 4 and 5, to specify the adapter control logic behaviour. Further, in section 5, together with the behaviour of two asynchronous ring adapters, we explain all the subsequent steps to get their implementations. After that, in section 6 we analyze and compare the performances of each of them. In the final section we assess all the obtained results to draw some conclusions.

2 Overview of pipeline ring protocol

The communication architecture of a Local Area Network (LAN) [8] with a ring topology is composed by a group of point-to-point interconnected nodes. As can be seen in figure 1, the user subsystem and its associated adapter can be identified in each node. The adapter allows the exchange of information with the rest of the connected nodes to the ring.

A well-known communication protocol of Medium Access Control (MAC) based on ring nets is the Token-Ring protocol. It is standardized by ANSI/IEEE as Standard 802.5 [2]. The Token-Ring was not aimed to be realised as an asynchronous design so, our adapter would not be fully compatible with the standard requirements. Nevertheless, we are going to use this protocol as a basic idea to build a scalable communication channel and to adjust it to the capabilities of asynchronous logic, free from any global clock.

In the Token-Ring, the network access of the message is coordinated by a circulating token. The token passes through the net enabling the injection of a message on its arrival to the node. So, a special pattern of bits has to be assigned to the token in order to be identified by the adapters.

In this type of access protocol to the ring we can distinguish two operation modes, the Busy protocol token-ring and the Lazy protocol token-ring [20]. In the former case the token is circulating permanently through the net, even when the net is empty of messages. In the latter case, the token travels through the net only when there is a request from one of the nodes to send a message in the net [20]. The Lazy protocol protocol consumes less power but offers the worse temporal performance (e.g., for a ring of 8 stations the average response time of the arbitration process is twice that of the Busy protocol) [20]. We are therefore going to employ the Busy token-ring protocol in our designs.

The use of the token-ring access assures two important properties: (1) that the first message that enters the net is the first one that exits it (FIFO discipline), and (2) that every messages have the same probability to access to the net. These properties are important because they help to resolve and in some cases, avoid the typical problems of message transmission in networks, the problems of *starvation* and *deadlock*. We say that the starvation problem occurs when some messages cannot be injected in the net for an indefinite time.

Using the token-ring protocol on a ring makes the net starvation-free since the number of nodes is finite, the size of each message is finite, and hence the token repeatedly reaches each node in a finite interval.

The deadlock problem happens when a message is blocked in the channel and cannot progress further. Using this protocol we can only imagine one situation where a deadlock may arise unless the system is properly implemented. Suppose a message has been injected by a node A, all the net resources are busy, no message is being consumed and therefore no resource can be released. In this situation there must be a message at the input channel of the injecting node A asking for a pass. According to the protocol, the message that the node A receives on the input channel is the one that has been injected by it. The protocol requires node A to consume this message (i.e., delete it from the ring). In order to avoid the deadlock problem the injection and the consumption process must be enabled *in parallel*.

Summarizing, we can state that the communication process in a ring topology with a FIFO injection and consumption protocol will be free of deadlock and starvation problems if every node that injects a message is able to consume the message that is arriving on its input ring channel. This fact has been checked empirically in subsection 4.4.

3 Modelling asynchronous circuits with Signal Transition Graphs

Successful results in the use of formal techniques and tools for synthesis and verification of asynchronous circuits have been recently achieved for labelled Petri nets, in particular for Signal Transition Graphs and Change Diagrams.

The model of Signal Transition Graph (STG) formalizes the behaviour of an asynchronous circuit and its environment, often represented by logic designers in the form of a timing diagram. An STG [9] is an interpreted Petri net in which the transitions are associated with up and down events of binary signals. STGs define the causal relationship between signal transitions, allowing deterministic and nondeterministic choice, synchronization points and concurrence.

In the STG, the events of the signals are represented with the signs $+$ and $-$. A transition of the STG labeled with $s+$ indicates an event of the rising edge of signal s , whereas $s-$ is a transition standing for the falling edge of s, and $s\tilde{}$ denotes any event (either rising or falling edge) on s.

The places of an STG that have only a predecessor and a successor transition are usually omitted in the graph and the m-tokens[1] in those places are marked directly as dots on the arcs. If a place has two or more predecessor or successor transitions it is represented explicitly as circle, and its marking with m-tokens is shown with dots put inside the circle.

We should point out that an STG describes a synthesizable circuit if it fulfils some Petri nets properties, like being free-choice, bounded (or sometimes 1-safe) and live, as well as some specific characteristics of the STG, like consistency and the property of Complete State Coding [9] [12]. Software tools, such as SIS, ASSASSIN, Versify and Petrify help the designer to guarantee those properties and obtain a correct implementation of an STG into asynchronous circuit.

Recently, a technique has been developed to synthesize STGs with internal conflicts between signals [4]. This method has been utilized to synthesize the control logic of our designs. The overall process incorporates an element of mutual exclusion (ME), often called Seitz' arbiter [17] to resolve the internal conflicts.

4 Behavioural description of a Token Ring protocol

The performance of the communication system depends strongly on the architecture and the operation of its adapters. For this reason, in this section we focus our attention on this module. The general operation of an asynchronous adapter is basically determined by the control logic, that will be detailed in sections 5.2 5.3 .

[1] We will use the word "m-token" for the tokens that mark STG places, to distinguish them from the token that circulates around the ring.

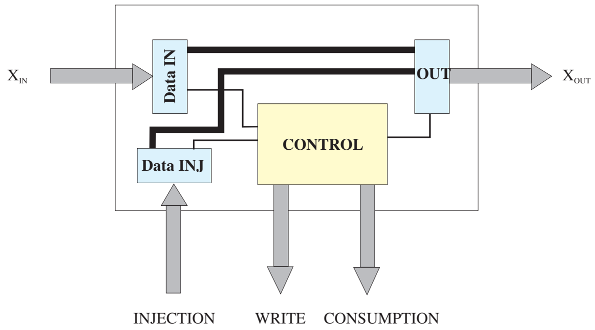


Figure 2: Adapter structure

4.1 Basics

The adapter is a hardware module that permits the exchange of information between a given node and the rest of the nodes in the ring. It carries out the following functions: message injection, consumption, passing and writing into the node.

The information to be transmitted in the net is divided into messages. Each message consists of the *head*, the *data* and the *tail*. The head indicates the destination address of the nodes that will read the data included in the message. The final part, tail, as well as the head, are assigned an unique code (bit pattern) such that the nodes can recognize the end and the beginning of a message, respectively.

Initially, when there are no messages in the net, the adapter is "on alert" for the token that circulates in the net. At any moment, if an user subsystem wants to transmit a message through the net, it will issue a request to its corresponding adapter waiting for the permission.

When the token arrives at a node two different operations can happen depending on the node state. The simplest case is when no message is waiting to be injected, so the adapter acts as a repeater, passing the message to the next node. In the other case, if a message is waiting to be transmitted, the token is retained inside the adapter and the injection of the new message is enabled. When the injection of the message is completed, the token is inserted into the net again. The injected message will cross all the ring nodes, being consumed or eliminated from the net by the same node that injected it.

4.2 General Structure

Regarding the internal logic of a token-ring adapter, we can distinguish the datapath and the control logic. Figure 2 shows an arrangement of the main blocks that compose the adapter besides its communication channels. With respect to the communication channels, each adapter connected to the ring has two channels, one for input, (X_{IN}) and the other for output, (X_{OUT}), which allow it to communicate with the rest of the ring. The $(INJECTION)$ and the $(CONSUMPTION)$ channels insert to and extract messages from the net. There is one more channel, $(WRITE)$, to allow the writing of the data inside the node.

Data-IN and *Data-INJ* are similar blocks. The *Data-IN* module detects the arrival of the new data item. In general, this module decodes the data, updates the head address (see below), informs the control logic of the token, head, data or tail arrival and encodes the new information.

The control logic is the most important component of the adapter and is synthesized from its STG description. We are going to describe in detail their behaviour in the following sections.

4.3 Node to node interface

When the data go through a communication channel from the sender to the receiver node, they suffer a variable delay which is greater than zero but finite. Then, when the signals are not local, it is necessary to use a communication protocol that guarantees a correct transmission. We must design a communication scheme for the sender and receiver that will be insensitive to variability in propagation delays. Hence, we are interested in *delay-insensitive communication* schemes.

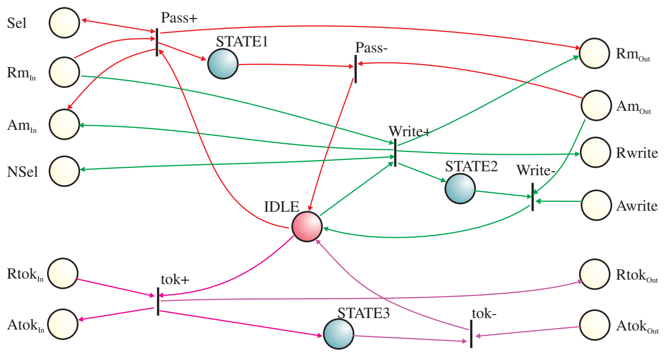


Figure 3: A simple STG of the adapter

The delay-insensitive codes are characterized as codes in which no code word[2] is included (in a set-theoretic sense) in another code word [19] [13]. This property is used by the receiver to decide without ambiguity when it has received a new code word and to generate an acknowledgement.

Among delay-insensitive codes, the most well-known are the One-Hot Code, Beger Code, Sperner Code and Knuth Code [19] [13] . In our case, the Sperner code has been chosen due to its property of being an optimally balanced code [18]. The latter means that the code cannot be extended with new code words of a given length without violating the delay-insensitive property and that there is no other delay-insensitive code of the same length with more words.

A code k-out-of-n is a code of length n composed of all the words of weight k. The size of a Sperner code k-out-of-n is $n!/(k!([n-k])!)$. For example, a code 6-out-of-3, C_6^3, (used in [21]), is formed by the words of length 6 with weight 3, e.g., 111000, 101100, 001011. In the same way the size of the code 6-out-of-3 is $6!/(3! * 3!) = 20$ words.

With respect to the channel width of the adapter, keeping in mind the complexity of the necessary logic and the transmitted information, we have decided (following the same argument as in [21]) to assign a width that allows us to transmit half a byte of information in one handshake beat.

Encoding half a byte needs 2^4 different code words for the data and four more code words to transmit the control signals. So for what is said above, we have used the code C_6^3, resulting in a channel width between two adjacent adapters to be 7 signals, 6 to transmit the Sperner-encoded data and one signal for the acknowledgement.

4.4 Petri Net Modelling of Protocol

In this section we are going to model the behaviour specification of the token-ring protocol using a high level STG model. Consequently, we can explore all the events that happen in the protocol execution for its verification. As a first step, abstracting the problem, let us reduce the adapter's function to the simpler case in which it acts as a repeater. The STG for this case is shown in figure 3.

When a message arrives in the input channel X_{IN}, an m-token appears in Rm_{In}. Then, taking into account the internal state and the external conditions, either the *Pass* or the *Write* path will be activated. Those external conditions are determined by the marking of the *Sel* and *NSel* places.

Then, the message advances until the output port Rm_{Out} and the adapter will remain either in state *STATE1* or *STATE2*, waiting for the acknowledgement signal. Next, upon the arrival of the acknowledgement, the adapter will return to the *IDLE* state and will be ready to receive a new datum.

The STG behaviour, when the adapter receives the token, is analogous to the one described above for the pass of a message. It is clear that this STG has no problem to be synthesized but we have not included the network access process. In other words, the STG must describe the injection and consumption processes. Figure 4 shows the adapter STG including all these processes.

Since the injection request may be produced by the user subsystem at any time, independently of the token arrival, a conflict may arise.

[2] A code word is an element of the code.

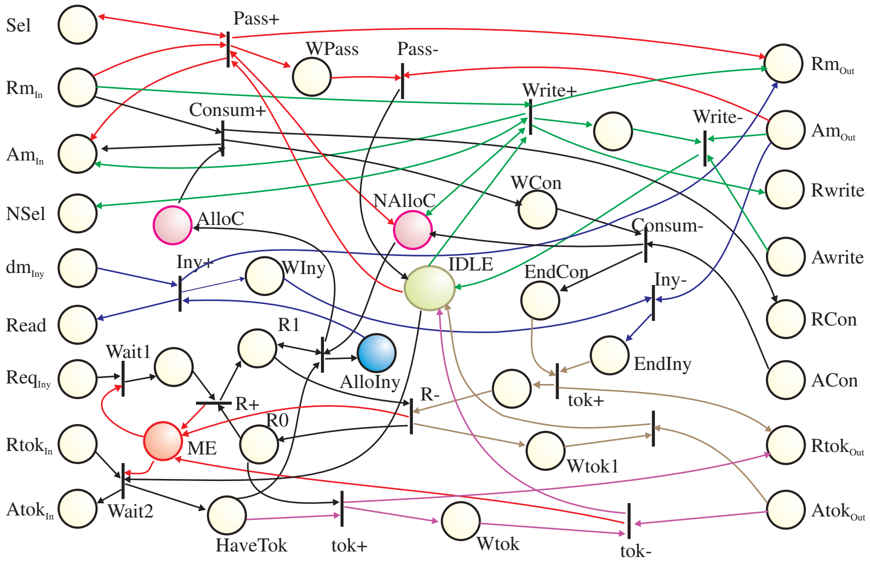


Figure 4: The STG of the adapter

The STG in figure 4 looks fairly complicated, and therefore it would be more appropriated to single out its main part involving conflict resolution into a separate model. Considering such a simplified version, with the injection process and the token propagation, as shown in figure 5-a, we see that the STG is not output-persistent with respect to $Atok_{in}+$ because in the marking (A, B, $< read-, Req_{Iny}+ >$) if $Req_{iny}+$ is fired $Atok_{in}+$ is disabled. The *output-persistence* property requires that none of the transitions of non-input signals are disabled by the firing of any other transition. In order to obtain an STG that can be synthesized by existing synthesis tools, e.g., SIS, we must guarantee this property [4]. Thus, the original STG must be transformed in such a way that it eliminates non-persistency on the original non-input signal transitions but, at the same time, preserves the original behaviour (in terms of traces of events generated by the model projected on the original set of events [4]).

This transformation consists basically of the introduction of a semaphore actions [4], as shown in figure 5-b. Each semaphore has an associated ME element whose implementation is shown in figure 5-c.

Even though the conflict was resolved including the ME element, the STG would not carry out the token-ring protocol since it would allow the user system to inject more than one message each time the token arrived. Therefore, the STG in figure 4 includes an internal state signal, R, that would prevent the injection of two or more messages.

When Req_{iny} wins the shared resource (the m-token in place *ME*) signal R changes its state and the resource is released. In this process, an m-token appears in *R1* indicating that a message is waiting to be injected upon the ring's token arrival. However, if $Rtok_{In}$ wins the arbitration (i.e. the ring token arrives before the user subsysetm's request), the resource will not be returned to place *ME* until the token has been sent back to the output channel, in which case no message can be injected into the channel.

The places *R0* and *R1* of the STG are mutually complementary, that is when one is marked with an m-token the other is empty. So, an m-token in *R0* means that the ring token can circulate.

With reference to the consumption process we should note that it is actived, $AlloC$, at the same time as the injection,*AlloIny*, so both processes can run in parallel (to avoid deadlocks, as mentioned earlier).

As a result, we have obtained an STG model of the adapter that can be synthesized into logic. Before we proceed to synthesis, we need to verify the protocol's STG. For this purpose several control modules have been assembled in a ring topology.

To make the overall system's model complete, we need to add STGs that describe the environment behaviour and the interconnection module. The latter module is an interface between the output and input channels of adjacent adapters that we have described above. This module generates the input signal Rm_{In}, according to the output signal Rm_{Out} of the previous adapter in the ring and decides at random about marking either Sel or $NSel$.

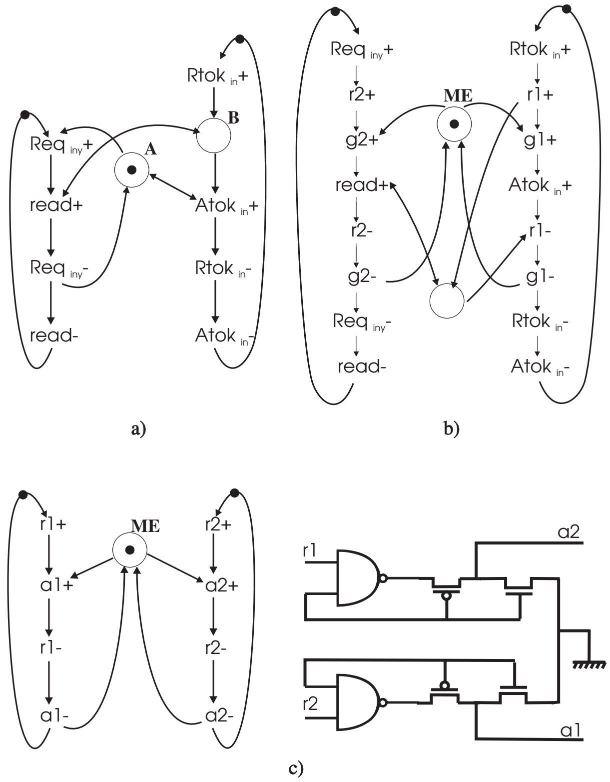


Figure 5: An output-persistent STG and use of Mutex elements

At this point, we are ready to verify the protocol behaviour. For that, we have built a ring with three adapters described as above. Using the Versify tool [14], the net model's behaviour has been analyzed and its has been found that the net is deadlock-free and starvation-free. So, our initial statement about the protocol's correctness, made in section 2, has been confirmed by means of such analysis.

5 Refining ring adaptor model

In this section we are going to derive circuits, both the control logic and the datapath, for adapters with different addressing schemes.

The control logic is the most important component of the adapter and is synthesized from its STG description. This logic works on the basis of a *four-phase handshake* protocol, also called a "return-to-zero" handshake. Initially, in this protocol, all signals are in the zero state. Then, the sender puts the data (encoded with six bits using the 3-out-of-6 code) on its output channel to the next node and waits for the ack signal to be set to logical one. When the latter event occurs, the data bits are returned to the spacer state (all zeros), which is followed by the return of the ack signal to its initial zero state.

The adapter datapath receives the code word through the inputs channels and produces the internal control signals to activate the control module.

5.1 Synthesis process

The control part of the token-ring adapter can be implemented in speed-independent logic, using synthesis and verification tools. This subsection overviews the overall synthesis process.

The asynchronous design tools need a mathematical model that reflects the asynchronous behaviour of the logic. Thus, the asynchronous speed-independent logic is based on the so called Muller's model of a circuit [18] that assumes an unbounded delay in the circuit's gates. An important characteristic of such circuits is that their operation relatively independent of the employed technology and environmental parameters, such as temperature.

The major steps in the design process are shown in figure 6. Initially, the control part behaviour is

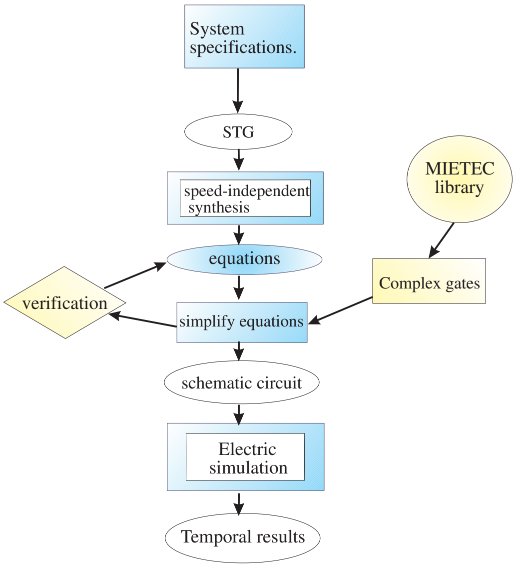


Figure 6: Synthesis method

specified and described as STGs. This must satisfy some properties in order to be implemented into a speed-independent circuit. As we mentioned above, the STG must be bounded, output-persistent and have a Complete State Coding to produce logic functions for its non-input signals. The logic equations can be obtained by SIS [10]. Some of these equations may however be overly complex to be physically realizable in gates available in the design library.

Thus, we decompose some *complex gates*, which have been assigned a complex logic function, into several gates with simpler logic expressions. In this decomposition we can use the following (informal) rule. A complex gate can be decomposed without introducing hazards if the output signals of the simpler gates are mutually orthogonal, that is, at the same time at most one of the output signals of those gates is set to logical one (sometimes called "one-hot" condition) [12]. For example, in figure 7-a we illustrate the decomposition of gate *abs* into the two mutually orthogonal signals *absMen* and *absTail*.

The equations obtained by such decomposition will still correspond to a speed-independent circuit. Nevertheless, owing to the fact that the manual decomposition process is not free of mistake, we can use the *Versify* tool to verify that the new equations fulfil the initial specification. So, at this point a feedback in the design process is possible.

Next, the resulting expressions are implemented at a schematic level using CMOS gates that carry out functions as $y = S + y\overline{R}$, where S and R are set and reset parts of the function. For instance, the gate represented in the figure 7-b is described by equation $y = a * b + y * (b + a)$, which is the logic function of a C-element, one of the main elements of the asynchronous logic. Such gates, in a more general form (called asymmetric C-gates), have been used extensively in the Amulet group's designs [1]; they are implemented in the MIETEC 2μ technology.

Once all the gates are designed, an electric simulation with hspice within FrameWorkII is done. This simulation allows us to check the correct operation of the adapter and obtain their time-performance characteristics.

The above-described synthesis process has been applied to the STGs of the ring adapter.

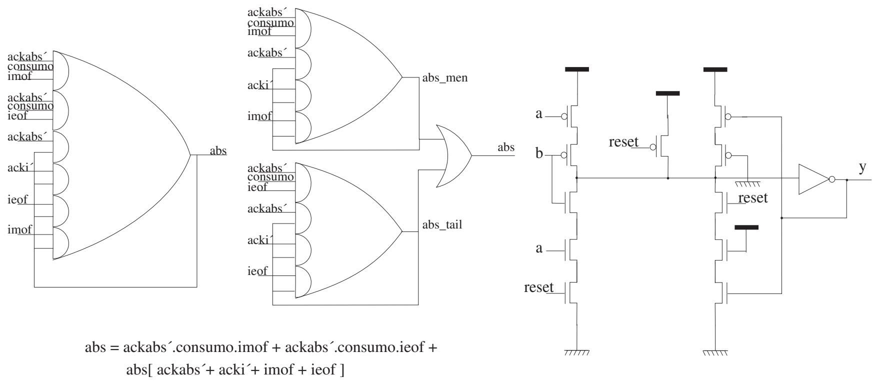
5.2 Deriving Circuits for a ring adapter: Case1

Knowing the adapter operation, some parameters, such as the channel width, the addressing, the interface between internal components must be fixed to get its implementation.

As we have said before, the channel width in this case is available to transmit half a byte of information using the code C_6^3.

The code C_6^3 has 20 different code words. 16 of them allow us to transmit half a byte of data and indicate the *relative address* to the destination node (see, e.g., [21] for a relative addressing scheme) when the flit[3] is

[3] A flit is the smallest unit on which flow control is performed



a) Complex gates decomposition. b) Gate implementation.

Figure 7: Complex Gates

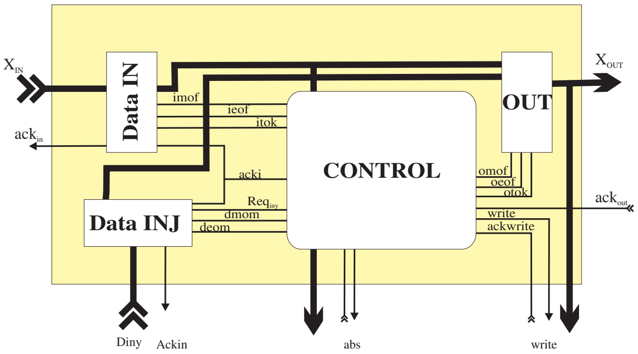


Figure 8: Input and Output signals of the control module.

the header. The other code words are used to transmit control information, such as the token or the tail of a message.

The principle of the relative addressing requires that the datapath of each node, when the message is passing it, updates the header address decrementing by one unit the destination address. As the flits of the message are encoded according to the C_6^3 code, we need to decode the address information, decrement the address and encode the new value. This process increases the latency of the adapter significantly. Trying to minimize the time to go through this block, we have mapped the code words with the address, avoiding decoding the data. That is, the decrementing is performed directly on the encoded data.

The speed-independent logic implementation of the control part is obtained from its behavioural STG description. This STG is quite complex, it includes a combination of behavioural paradigms of free-choice, arbitration-choice and parallel processes.

5.2.1 Interface

Every delay-insensitive code word transmitted between a pair of adapters in the ring is detected by the input datapath *Data IN* or *Data Inj* of the receiver and generates events on the input control signals. If the token arrives, the signal *itok (input token)* is raised, but if the new flit is a datum, either the signal *imof (input-medium-of-flit)* or *ieof (input-end-of-flit)* is produced, distinguishing the tail from the rest of the message. All these signals of the input channel are requests and the control module replies with an acknowledgement on *acki* to complete the four-phase handshake protocol. Figure 8 highlights the most important interconnections between modules.

The signals of the output channel *otok, omof, oeof* and *acko* have a meaning similar to *itok, imof, ieof*

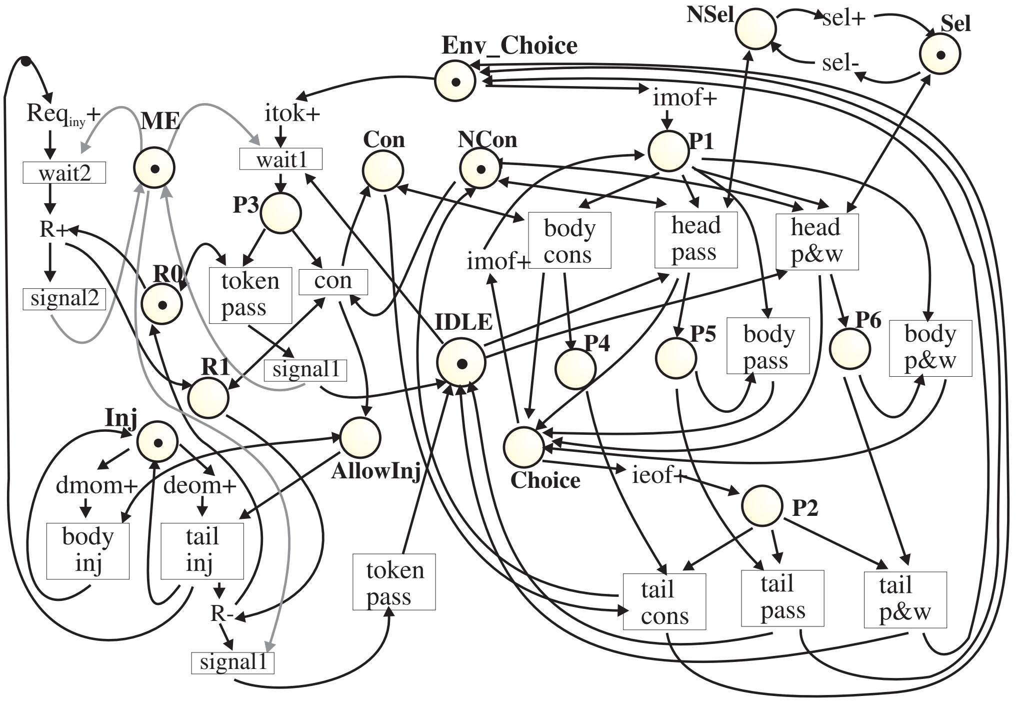


Figure 9: STG for the case 1 adapter

and *acki*. The signals *Req, dmom* and *deom* are related with the injection of a message. They stand for the injection request, and data in the middle and at the end of a message, respectively. *Write* and *Ackwrite* are associated with the writing of data items into the node, whereas *Abs* and *AckAbs* with the consumption of message data without writing them.

5.2.2 Formal model of control logic

Due to the complexity of the STG only the most important parts will be commented. In figure 9 a global scheme of the STG is introduced. It has several free-choice places like *Env_Choice*, *Choice* and *Inj*, whose firing relies on the environmental signal state. *Iny* is associated with events that occur in the injection channel. *Env_Choice* and *Choice* are related to events in the input ring connection channel.

In a message we can distinguish the head flit, that opens the path, the body and the tail of the message, that returns the circuits to the initial state. The head of the message contains the destination address that specifies the node where the message must be written.

When an event arrives on *imof*, the level of input signal *sel* (a node selection flag) when the node is in the IDLE state determines whether the message is to be written into the node or passed further. The selection flag is stored inside the adapter in a state variable until the tail is received.

Once the pass or the writing of a message is activated the only parallel process that can happen is the arrival of an injection request. This request changes the internal state of the logic by means of recording this fact into the internal variable R . The high value of R will initiate the injection of a message when the token arrives.

Due to the conflict between the injection request and the token arrival we need to incorporate an arbitration element making the graph output-persistent. The STG allows alternative actions depending on the resolution of the race between Req_{Iny} and *itok*. If Req_{Iny} wins the ME, the R signal is toggled to 1, after which the ME is released. However, if *itok* arrives in the adapter and wins the ME, depending on the value of R, two different events can happen:

– if $R = 0$, signal *otok* is produced, releasing the ME element and putting the logic into the IDLE state.

– if $R = 1$, which means that there is a message waiting to be injected, the token remains in the adapter, the injection is started and the consumption of the next arriving message is allowed. The token is released after the injection of the message tail, which brings the STG model into the IDLE state.

The complete STG model of control has 11 input signals, 9 output signals and 5 internal state signals

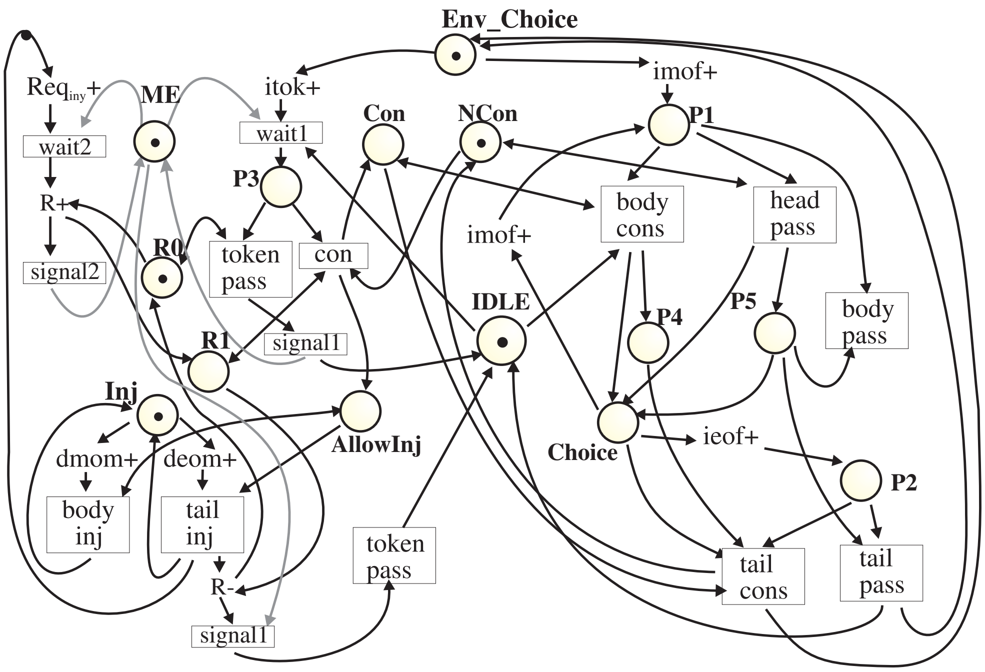


Figure 10: STG for the case 2 adapter

insuring that the STG has a complete state coding. We should point out that the synthesis tool (SIS) is very close to its power limit with an STG of that size.

5.3 Deriving Circuits for a ring adapter: Case2

In the communication processes the physical support, that is, the communication medium is usually a bottleneck. It influences a lot the final latency of the message. Our motivation for Case 2 is to minimise the latency compared to Case 1. This can be achieved by reducing the functionality of the adapter, and hence the delay of its logic, to the minimum.

The internal structure of the adapter remains similar to the previous case but its operation is modified so as to allow the message to go through the logic as fast as possible (cf. the idea of “mad postman” routing [22]). The main difference with Case 1 is that in Case 2 every flit that passes through the adapter is unconditionally written into the node (which may sort out the address selection problem at a higher level). This would reduce the latency as the input datapath logic does not have to process the destination address as a part of the overall critical delay path. This approach is similar to the idea of Scalable Coherent Interface (e.g. Ramlink) [5]. Thus the functions of the adapter are: injection, consumption and pass with writing.

The message structure is the same as in the previous case: head, body and tail, but now the destination address is absolute. Each node of the ring has an unique identifier of its position in the ring. The higher level protocol ensures that the address is compared with the positional identifier of the node. If they are equal, the message goes into the node. Otherwise the message is dropped. Those actions are realised inside the User Subsystem of the node and are not part of the main ring delay path.

5.3.1 Formal model of control logic

The input and output signals of the control module are the same as in the previous case but the *sel* signal is missing because it is no longer necessary. The STG specification of the control module is simpler than in Case 1. Figure 10 shows a simplified version of the STG. Basically, we can distinguish the following states: IDLE when no action takes place, pass of the token, injection of a message, enabling the consumption and pass of the message together with writing it in the node. The injection and consumption request have the same behaviour as in the Case 1, described in section 5.2

This STG has 11 input signals, 9 output signals and 3 internal signals that can be synthesized by synthesis tools.

6 Performance of the ring channel

Having completed the design of the communication interface, we have obtained its temporal parameters simulating each of the two adapter designs with HSPICE. The measured times have been used to determine the net performance in terms of the average latency of a message.

The latency of a message, equation 1, is the time that the message spends in the interconnection network, excluding the overhead of injecting it into the network and pulling it out when it arrives. Then, the latency of a message, according with [7], is the time of flight plus the transmission time. The time of flight is the time for the first flit of the message to arrive at the receiver, including the delays due to the hardware. The transmission time is the time for the message to pass through the net; it is equal to the size of the message divided by the bandwidth. This measure assumes there are no other messages to contend for the network.

$$Latency = Cable\ delay + Adapter\ latency + \frac{Message\ size}{Bandwidth} \tag{1}$$

6.1 Verification at circuit level

The main time characteristics of the adapter have been obtained by electrical simulation with HSPICE. In table 1 we show the propagation time of some control signals for the adapter described in the section 5.2, the one in which the message could be written in the node or simply to pass through it. The propagation time for the signals depends on the actual state of the logic. For example, the time from an event in *imof* occurs until *omof* toggle is higher if the flit is written in the node than if it only passes. The subscript *write* will appear in the table 1 if the time makes mention to the delay when the flit is written in the node.

We should pount out that we can obtain different delays in the propagation of some signals depending on whether the head or data is transmitted. In table 1 we have reflected those times in the columns called *head* and *data*, respectively.

Signals	**time (ns)**		**Signals**	**time (ns)**	
	head	**data**		**head**	**data**
$D_{in}, imof$	6.73	3.98	$imof, omof_{write}$	11.13	7.9
$D_{in}, itok$	5.97	2.81	$imof, omof$	6.66	3.78
$D_{in}, ieof$	2.52		$imof, acki_{write}$	14.24	10.76
D_{inj}, Req	4.24		$imof, acki$	9.55	6.65
$D_{inj}, dmom$	2.55		$ieof, oeof_{write}$	9.5	
$D_{inj}, deom$	1.22		$ieof, oeof$	6.98	
$itok, otok$	9.84		$ieof, acki_{write}$	12.49	
$itok, acki$	10.46		$ieof, acki$	9.84	

Table 1: Time for the adapter *Case 1*

The values of the same parameters are shown in table 2 for the adapter described in section 5.3.

As we can see the propagation time of the datapath for Case 2 has decreased, affecting the final communication performance. In general, the propagation time for the control signals is minor than for the Case 1 when the flit is written in the node but higher than when the flit passes through the adapter.

6.2 Latency of the message: Case 1

The latency of a message is expressed as the sum of the time it takes to pass the head of the message through all the ring adapters plus the time for each one of the remaining flits of the message to be consumed. The head time to pass through the adapters with the relative addressing mechanism, described in section 5.2, is given by equation 2.

$$Latency_{head} = T_{inj} + (N-2)T_{pass} + T_{write} + T_{con} \tag{2}$$

N is the number of adapters in the ring.

Signals	time(ns)	Signals	time(ns)	
			head	data
$D_{in}, imof$	2.35	$imof, omof_{write}$	8.47	5.44
$D_{in}, itok$	1.75			
$D_{in}, ieof$	1.42	$imof, acki_{write}$	11.36	8.35
D_{inj}, Req	4.24			
$D_{inj}, dmom$	2.55	$ieof, oeof_{write}$	7.50	
$D_{inj}, deom$	1.22			
$itok, otok$	10.91	$ieof, acki_{write}$	10.34	
$itok, acki$	14.57			

Table 2: Time for the adapter *Case 2*

The first term of equation 2, T_{inj}, represents the time since the token arrives until the request to pass the head to the following adapter is received. T_{pass} is the time to pass through the head. T_{write} is the time it takes to write information into the node. Finally, T_{con} is the time it takes to consume the head in the node which injected the message.

Filling each of these terms with the corresponding values from table 1, we obtain an expression for the head latency shown in equation 3.

$$Latency_{head} = 7.02 + 17.49 * N \; ns \tag{3}$$

Then, for a ring with $N = 16$, the latency of the head is 287 ns.

Concerning the total latency of the message, it is obtained as the sum of the head latency plus the time to consume the rest of the flits that compose the message (equation 5).

$$\begin{aligned} Latency &= Latency_{head} + (L-1)\, T_{conflit} \qquad (4) \\ &= 53.83 + 17.49 * N + 44.13(L-2) \; ns \qquad (5) \end{aligned}$$

This expression is a function of the number of adapters in the ring, N, and the length of the message, L. Then, for a ring with $N = 16$ and the message length equal to 32 flits, the latency is 1658 ns.

The previous expressions show the latency of a message in a ring when injection is performed in only one of the adapters, and therefore, only one message is travelling in the network.

The increase of the applied load in the ring will increase the latency of the messages as a function of the number of flits of each previously injected message. We assume that the token is held inside the adapter injecting the message until the injection is complete. During such a token tenure of one of the adapters other potential message injectors must be in a waiting state.

6.3 Latency of the message: Case 2

The expression of the latency of a message that travels through a LAN ring consisting of the Case 2 adapters (section 5.3) is similar to equation 2. However, in this case, the time of passing the head is always the same, since every datum that crosses the adapter is written in the node. The numerical values obtained for this case are shown in equation 6.

$$Latency_{head} = 0.78 + 14.92 * N \; ns \tag{6}$$

For example, the head latency for a ring with 16 adapters is 239 ns.

The overall message latency can be found in an analogous way – as the sum of the time used by the head flit passing through the ring plus the consumption time of each flit of the message. So, substituting the numerical values for each term we obtain the latency as a function of N and L (equation 7).

$$Latency = 44.26 + 14.92 * N + 33.33(L-2) \; ns \tag{7}$$

For a 32 flit long message in a ring formed by 16 adapters, the latency of the message will be 1283 ns. Comparing this result with the latency for Case 1, we can note about 25-30% decrease in the time necessary to send information between different nodes of the ring. For long messages this difference is obviously determined by the ratio between the bandwidths, i.e. by the time of passing/consuming a flit, 44 ns for Case 1 versus 33 ns in Case 2. The faster version's bandwidth is 121Mbit/s (recall that one flit carries 4 bits). This figure can be further improved by optimising the STG specification – e.g., allowing more concurrency between writing in the node and passing the data forward. More concurrency may however cause Complete State Coding problems, and due to the present limits on the size of STGs, the existing tools would have difficulty coping with such STGs. This is a good motivation for their further enhancement. Another possibility could be to avoid sending acknowledgements for every flit between adjacent adapters, relying on realistic timing constraints. Significant improvements have been reported in such FIFO designs [23].

7 Conclusions

An asynchronous design of a communication interface adapter has been presented in this paper. The adapters work in a ring structure using a token ring protocol, whose main principles of message injection and extraction are similar to the "standard" Token Ring [2]. Unlike the classical synchronous design of the Token Ring, our inter-adapter communication uses a four-handshake asynchronous protocol. The logic implementation of the adapters has reflected the potential trade-offs in organising the addressing mechanism. Two different approaches have been considered. The first approach is the one in which the adapter function includes the decoding of the destination address and maintaining the selection flag. The second approach "postpones" the address processing until the message reaches the user subsystem. It simply copies each datum in the incoming message from its input channel to both output channel and its internal node. The first method relies on the so-called relative addressing technique, originally described in [21], where each adapter checks the most significant bit in the address and shifts the address before passing it to the next adapter in the ring. Since the data elements (including the address bits) must be encoded with a delay-insensitive code (we used a Sperner code "3-out-of-6"), the logic performing the above address processing is not as trivial as with the "normal" positional coding, which affects negatively the overall message latency. Within the second approach, the datapath delay is minimised as no address processing is done. The negative consequence is however that this function is now on the higher level protocol logic. The typical gain in latency due to the second method is not extremely impressive however, only about 25-30%. This figure can be improved by, e.g., further optimising the STG models and allowing interaction between adjacent nodes without acknowledging every flit.

Our major effort has been put in developing implementable STG models and obtaining the speed-independent control logic for the two design options of the ring adapter. The synthesis tool that has been mainly used for that is SIS [10]. HSPICE simulation has allowed us to obtain the adapter's temporal characteristics and finally estimate latency of the message transfer in the ring.

In the process of logic synthesis with SIS, we have approached its capability limits, such as the total number of input, output and internal signals of a complete STG should not be greater than 30. Also for a strongly coupled specification, the equations that describe the output behaviour, have complex products with more than 5 or 6 signals. Using CMOS NAND gate with more than 5 inputs may result in a large delay, hence an automatic decomposition method is needed. So far we have sometimes used manual decomposition based on the idea of orthogonal ("one-hot") operation of complex gates in different mutually exclusive modes. Splitting them into two or more gates together with an OR gate appears to be safe but for complete assurance of speed-independence, verification (e.g., using Versify) is required. In the future, tools like Petrify [3] are going to perform logic decomposition and technology mapping in a much more efficient way.

Acknowledgements

The authors would like to thank Jordi Cortadella from Polytechnic University of Catalonia, Barcelona, for his ideas about designing the Extract-Insert type of communication interface, and his help in arranging Carmen Carrion's visit to Newcastle, and Ramon Beivide Palacio, Carmen Carrion's PhD supervisor at Santander, for supporting and guiding her research. Many thanks also go to Luciano Lavagno (Torino/Berkeley), Oriol Roig (Barcelona) and Alex Semenov (Newcastle) for their kind help with their software tools, SIS, Versify and PUNT.

References

[1] Amulet Group – Low Power Design. *Cell Library Datasheets.* Dept. of Computer Science, 1995.

[2] ANSI/IEEE Standard 802.5 Working Group. *Token Ring Access Method and Physical Layer Specifications.* IEEE, N.Y., 1985.

[3] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *Proc. of the 11th Conf. Design of Integrated Circuits and Systems*, Barcelona, Spain, Nov. 1996, to appear.

[4] J. Cortadella, L. Lavagno, P. Vanbekbergen, A. Yakovlev Designing Asynchronous Circuits from Behavioural Specifications with Internal Conflicts. *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, SaltLake City, Nov. 1994, IEEE Comp. Society Press, N.Y., pp. 106–115.

[5] D.B. Gustavson and Q. Li. Local-area multiprocessor: the scalable coherent interface. *SCIzzL Web-Server*, posted by special arrangement with Bit 3 Computer Corp., 1995.

[6] S. Hauck. Asynchronous Design Methodologies: An overview. *Proceedings of IEEE*, vol 83, No 1, pp. 69–95, January 1995.

[7] J. Hennessy and D. Patterson. *Computer Architecture: A quantitative approach.* Ed. Morgan Kaufmann, Second Edition 1996.

[8] A. Hopper, S. Temple, R. Williamson. *Local area network design.* Addison-Wesley Intern. Comp. Sci. Series, 1986.

[9] L. Lavagno. *Synthesis and Testing of Bounded Wire Delay Asynchronous Circuits from Signal Transition Graphs.* Ph.D. thesis, University of California at Berkeley, 1992.

[10] L. Lavagno, E.M. Sentovich, K.J. Singh, C. Moon. *SIS: A system for Sequential Circuit Synthesis.*. Memorandum No. UCB/ERL M/92/41 Electronics Research Laboratory, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94729, May 1992.

[11] B. Lin, C. Ykman-Couvreur, H. De Man *Assassin: A synthesis system for asynchronous control circuits.* Technical Report, IMEC, Sept 1994, User and Tutorial Manual.

[12] E. Pastor. *Structural Methods for the Synthesis of Asynchronous Circuits from Signal Transition Graphs.* Ph.D. thesis, Universidad Politecnica de Catalunya, Febrero 1996.

[13] P. Patra, D.S. Fussell. *Power-efficient Delay-insensitive Codes for Data Transmission.* Dept. of computer Sciences, University of Texas. Available on net.

[14] O.Roig, J. Cortadella. *Versify manual.* Dept. of Computer Architecture, Universitat Politecnica de Catalunya, 1995.

[15] P.T. Roine. A system for asynchronous high-speed chip to chip communication. *Proc. Sec. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems,* Aizu-Wakamatsu, Japan, March, 1996, pp. 2-10.

[16] C.L. Seitz. System timing *Chapter 7 of Introduction to VLSI Systems*, C. Mead and L. Conway, Addison-Wesley, 1980.

[17] C.L. Seitz. *Ideas about arbiters.* Lambda, Vol. 1, First Quarter, pp: 1-14, 1980.

[18] V.I. Varshavsky, M.K. Kishinevsky, V.B. Marakhovsky, V.A. Peschansky, L.Ya Rosenblum, A.R. Taubin and B.S. Tsirlin. *Self-Timed Control of Concurrent Processes.* V.I. Varshavsky (ed.), Kluwer Acad. Publ., Dordrecht, 1990 (Translated from Russian; Russian Edition – Nauka, Moscow, 1986).

[19] T. Verhoeff Delay-insensitive codes: an overview *Distributed Computing*, vol. 3, no. 1, pp: 1-8, 1988.

[20] K.S. Low and A. Yakovlev *Token Ring Arbiters: An Exercise in Asynchronous Logic Design with Petri Nets.* Technical Report No. 537, University of Newcastle upon Tyne, UK, November 1995.

[21] A. Yakovlev, V.Varshavsky, V.Marakhovsky and A.Semenov. Designing an asynchronous pipeline token ring interface. *Proc. of 2nd Working Conference on Asynchronous Design Methodologies*, London, May 1995, IEEE Comp. Society Press, N.Y., 1995, pp. 32-41.

[22] J.T. Yantchev, C.R. Jesshope. Adaptive, low latency, deadlock-free packet routing for processor networks. *IEE Proceedings-E*, May 1989.

[23] J.T. Yantchev, C.G. Huang, M.B. Josephs and I.M. Nedelchev. Low-latency asynchronous FIFO buffers. *Proc. of 2nd Working Conference on Asynchronous Design Methodologies*, London, May 1995, IEEE Comp. Society Press, N.Y., 1995, pp. 32-41.