Design and Analysis of a Self-Timed Duplex Communication System

Alex Yakovlev, *Member*, *IEEE*, Steve Furber, *Senior Member*, *IEEE*, René Krenz, *Member*, *IEEE*, and Alexandre Bystrov, *Member*, *IEEE*

Abstract—Communication-centric design is a key paradigm for systems-on-chips (SoCs), where most computing blocks are predesigned IP cores. Due to the problems with distributing a clock across a large die, future system designs will be more asynchronous or self-timed. For portable, battery-run applications, power and pin efficiency is an important property of a communication system where the cost of a signal transition on a global interconnect is much greater than for internal wires in logic blocks. The paper addresses this issue by designing an asynchronous communication system aimed at power and pin efficiency. Another important issue of SoC design is design productivity. It demands new methods and tools, particularly for designing communication protocols and interconnects. The design of a self-timed communication system is approached employing formal techniques supported by verification and synthesis tools. The protocol is formally specified and verified with respect to deadlock-freedom and delay-insensitivity using a Petri-net-based model-checking tool. A protocol controller has been synthesized by a direct mapping of the Petri net model derived from the protocol specification. The logic implementation was analyzed using the Cadence toolkit. The results of SPICE simulation show the advantages of the direct mapping method compared to logic synthesis.

Index Terms—Asynchronous circuits, communication protocols, modeling, Petri nets, power and pin efficiency, self-timed circuits, signal transition graphs, synthesis.

✦

1 Introduction

The International Technology Roadmap for Semiconductors (ITRS 2001) predicts that, due to the increasing problems with distributing a clock across a large die, future designs will be more asynchronous. Self-timed circuits offer a number of advantages for system design, namely, low power consumption, electromagnetic compatibility, greater modularity, and operational robustness. One area of digital design where circuits with global asynchrony are seen more as an inevitable technological reality rather than an optional design discipline is interfacing. Development of formally sound methods and tools to support design of communication protocols and interfaces for systems-on-chip and multichip systems is a difficult problem. The best way to tackle such a problem in its generic form would be to attempt an interface design example maximally using formal techniques. This way is also motivated by the fact that most future SoC designs will be communication-driven [2].

There are many ways that information can be communicated between chips, but a system with a delay-insensitive (DI) encoding of the transmitted data may be preferred because it is completely robust to variations of delays associated with wires [21]. For example, in a DI N-of-M code, every valid code combination of M bits must have exactly N bits equal to logical-1. There is an additional combination, e.g., all-zeros, which is called a spacer. In data transmission with return-to-zero (level-based) signaling, each valid combination must be followed by a spacer. The key property of an N-of-M code, guaranteeing its delay-insensitivity, is that every transition from a spacer to a valid combination never passes through another valid combination. Examples of N-of-M codes, e.g., 1-of-4, 3-of-6, 2-of-7, have been used in self-timed systems described in [22], [1].

An interchip communication system with an N-of-M encoding and a possibility of duplex interaction, in which both communicating agents can transmit their data and at the same time acknowledge the receipt of the other party's data has been proposed in [9]. The use of an N-of-M code is also combined with a non-return-to-zero (transition-based) signaling, in which logical-1 and logical-0 bits are associated with transitions and their absence, respectively, as opposed to the return-to-zero signaling which would require a spacer. This system optimizes both power and pin efficiency in comparison with the system consisting of two simplex handshake channels. The overall view of the system with the agents being called Master and Slave is shown in Fig. 1. The construction of a communication controller for this system presents an interesting case study for asynchronous design techniques. This process is described in this paper.

First, a formal description of the protocol is constructed. According to the widely accepted notion, a protocol is a precise set of rules governing the behavior of the communicating agents. In this paper, the protocol is defined using the concept of a *protocol state machine* shown in Fig. 2a. It is

- *A. Yakovlev and A. Bystrov are with the School of Electrical, Electronic and Computer Engineering, University of Newcastle, Newcastle upon Tyne NE1 7RU, U.K. E-mail: {alex.yakovlev, a.bystrov}@ncl.ac.uk.*
- *S. Furber is with the Department of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, U.K. E-mail: sfurber@cs.man.ac.uk.*
- *R. Krenz LECS-IMIT-KTH, Electrum 229, 164 40 Kista, Sweden. E-mail: rene@imit.kth.se.*

Manuscript received 20 Aug. 2002; revised 2 July 2003; accepted 18 Sept. 2003.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 117154.

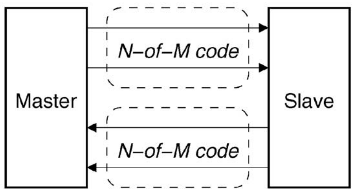

Fig. 1. Overall communication system.

based on the principle of information hiding, which is crucial for a compact and unambiguous protocol definition [5]. The protocol state machine fully abstracts away actions performed by each agent internally, and only shows actions that *may occur* in the interface between the agents.

Second, the communication protocol is formally verified with respect to its requirement to be deadlock-free and provide delay-insensitive communication. For this, another communication model is constructed from the main protocol machine following the approach from [5]. This global model is schematically shown in Fig. 2b, includes separate models of the agents, Master and Slave, together with the models of the communication channels and the links with the protocol users (source and destination processes). Here again, only minimum necessary information is included into the system model to hide any irrelevant internal activity in both parts of the system. As the overall model requires capturing concurrency, choice and arbitration, Petri nets have been chosen to represent its behavior [13]. The use of Petri nets offers a possibility of efficient formal check of the model. Petri nets capture concurrency in its natural (causal) form; therefore, model-checking can employ partial order techniques, avoiding the painstaking process of generating the full state-space explicitly. An existing analysis tool based on Petri net unfolding has been used to prove the deadlock-freedom and delay-insensitivity of the protocol [16].

Our third task is to design control logic and, thus, implement the protocol using standard CMOS components in order to estimate the performance of the communication scheme and assess a number of trade offs. For example, it is interesting to determine the penalty in terms of performance for the gain in power and pin efficiency offered by this duplex scheme compared to a pair of unidirectional handshakes. Another objective is to compare different ways of deriving asynchronous control logic from the Petri net specification of the controller.

The implementation section illustrates the process of constructing a Petri net specification for control logic in Master and Slave. The Petri net model of each controller is then translated to a circuit implementation using a method of *direct mapping* [23]. The method associates a memory latch, called a David cell (named after René David, who proposed an elegant way of one-hot implementation of asynchronous FSMs with parallel actions, cf. [6]), with each place in the Petri net. The transitions of the Petri net model are labeled with the names of operations or commands that the main controller issues to the send and receive interfaces responsible for the transmission and reception of the dual-rail data into and from the m-to-s and s-to-m channels. The direct mapping approach has the advantages of modularity and structural transparency between the behavioral model and the logic structure. It avoids refining the relatively high level Petri net specification of the controller into a more detailed form, such as Signal Transition Graph (STG) [15] or a Burst-Mode State Machine (BMSM) [14]. Although both the STG-based and BMSM-based modeling techniques and associated logic synthesis tools, Petrify and Minimalist, could do the job of implementing the controller specifications, the complexity of model refinement and logic synthesis would have been significantly greater.

The main controllers in Master and Slave obtained by direct mapping are subsequently composed with the send and receive interfaces. The latter perform protocol actions as commanded by the controller and in accordance with the required N-of-M encoding of the channel data. In this paper, in order to keep the encoding and decoding logic simple, we consider a 1-of-2 (dual-rail) encoding and non-return-to-zero signaling for power efficiency. The composition of the controller and the interfaces is very simple. It is based on request and acknowledgement handshakes associated with the commands issued by the main controller. Finally, the performance of the overall system is studied by SPICE simulation and compared to the system of two unidirectional handshake channels.

The use of direct mapping from a Petri net specification is a distinct feature of our design methodology, which produces results that compare favorably against circuits obtained by logic synthesis from the Petrify tool [4]. This design example can be seen as an important benchmark for future automatic control synthesis tools employing direct translation techniques [3], [17]. The overall methodology based on formal specification of a protocol, its refinement as a controller specification and, finally, direct mapping to logic, guarantees correctness of the design described in this paper.

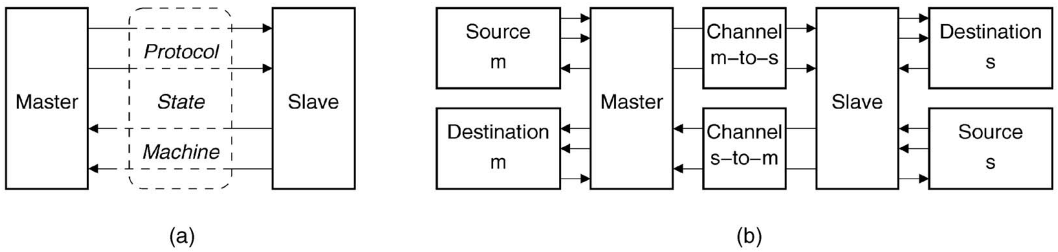

Fig. 2. Basic structures for protocol specification and verification: (a) Protocol state machine and (b) communication system with users and channels.

2 Asynchronous Communication System and Its Protocol

2.1 Bidirectional Communication Scheme

The classic non-return-to-zero (NRZ) dual-rail approach with acknowledgement requires at least six wires in order to provide a bidirectional communication channel. Each direction uses two wires to transmit one bit of data (by making a transition on wire 0 when transmitting a logical-0 and a transition on wire 1 when transmitting a logical-1) and one wire to transmit an acknowledgement. Using other types of DI codes, e.g., N-of-M codes, is also possible; this may increase the pin-efficiency compared to the dual-rail code but at the cost of having additional logic to convert the N-of-M code to normal one and vice versa.

The scheme proposed in [9] optimized the classic dual-rail mechanism by using *only four* wires, two in each direction, and exploiting the data wires in one direction to carry acknowledges for communication in the other direction. Namely:

- Four wires are used for bidirectional communication. A0 and A1 carry data symbols in one direction; B0 and B1 carry data symbols in the other direction.
- During true bidirectional communication a transition on A0 or A1 is acknowledged by a transition on B0 or B1, and vice versa.
- During unidirectional communication the same protocol applies, but the returned data is void. The returned value is called an "Ack" symbol.
- The start of valid data is indicated by preceding it with a "Start" symbol which is acknowledged by an "Ack" symbol. For example, an "Ack" symbol response could be a zero and a "Start" symbol a one. A predefined number of bits following a "Start" symbol represent valid data.

For a low-power communication system, it is desirable for us to minimize the number of transitions on wires used to send a given data value. In particular, we want to avoid sending transitions when there is no data to send. The above bidirectional discipline has therefore been enhanced with a mechanism in which either end of the channel may initiate communication at any time, including the possibility of both ends to initiate it at the same time (to within some tolerance). The latter condition is called collision. Initiating communication requires the generation of a Start symbol. In order to resolve collision in the situation when two Start symbols have been generated independently by both ends, the following is ensured in the protocol:

- Both ends of the channel *know* there is a collision in the system as both will issue a "Start" symbol and receive a "Start" symbol instead of an Ack symbol.
- One end of the channel must *defer* to the other. The end which defers is called the Slave and the other the Master.
- The Slave defers by retracting its "Start" symbol and replacing it by an "Ack."

In a true DI system, true retraction is not possible since, once the sender has made a transition on a wire, it cannot make another transition on the same wire until it has had confirmation (in the form of some sort of acknowledgement) that the first transition has been received at the other end. Instead of such a single wire retraction, which is delay-dependent, a special symbol, called "SlaveAck," is used. SlaveAck subsumes the Start symbol. In the case of dual-rail encoding, a SlaveAck symbol is superposition (i.e., union) of Start and Ack symbols. Thus, the wire that makes a transition in the Start symbol also does so in the SlaveAck symbol, and the additional wire which does not make a transition in the Start symbol makes a transition within the SlaveAck symbol to indicate a retraction of the Start symbol. That second wire is also used, on its own, to represent an Ack symbol.

To sum up, in a dual-rail DI system, 01 can be used for Start, 10 for Ack and 11 for SlaveAck.

2.2 The Protocol

The full protocol for the above communication scheme is represented by a state-transition graph in Fig. 3. This graph depicts an imaginary (protocol) state machine, placed between Master (M) and Slave (S), which defines permissible signal sequences on the wires connecting Master and Slave. Serving the purposes of the protocol definition only (cf. [5]), this machine does not show any behavior that is internal to Master and Slave. The six major states are Idle, Slave transmit (Ts), Master transmit (Tm), Retract (Ret), and two duplex states (TsTm and TmTs). These states together with minor (transient) states, shown by small circles, are assigned to four main modes that are Initialization (I), two simplex modes, Master transmit (II), Slave transmit (III), and one duplex mode (IV).

For every state in the protocol, certain transmission events are allowed to take place. For example, when the protocol is in state Idle, there may either be received a Start symbol from Slave (S/Start) or a Start symbol from Master (M/Start). Likewise, when in state Ts, either "S/Data/Last" or "S/Data/NotLast" symbol may be received from Slave. The notation "S/Data/Last" indicates that Slave sends its last data value.

The labeling of these arcs, e.g., m01 or s10, is associated with the source of the signal ("m" for Master and "s" for Slave) and the dual-rail encoding of the symbols (Start, Ack, and bit-data values) and its interpretation, whether it is a special symbol or a bit value, depends on the current mode and the state. Thus, s10 (s01) indicates that a transition is made on wire 0 (1) that comes from Slave. Similarly, m10 (m01) stands for a transition on wire 0 (1) from Master. When occurring in (coming out or leading into) the initialization mode, these labels correspond to symbols; in simplex transmission modes, they are used for a data value from one side and a symbol from the other side; finally, in the duplex mode, these labels always correspond to data values only.

The protocol implicitly has two types of choice in its states. One such type, nonarbitrating choice, is made in states Ts, Tm, TsTm, and TmTs. It depends on whether the transmitted value is last or not. It is assumed, for simplicity of logic in the controllers, that both the sender and receiver have the inner (higher level) facilities for counting the number of data values transmitted and received in such a

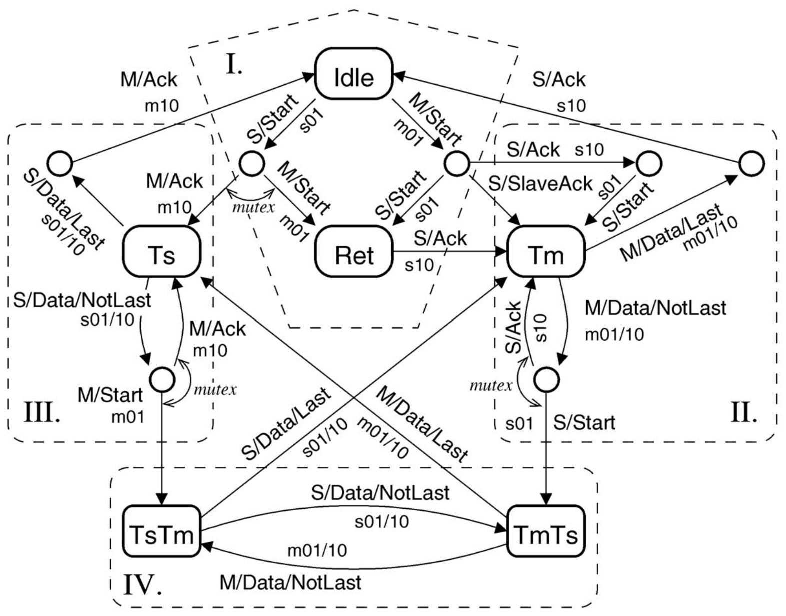

Fig. 3. Protocol state diagram.

way that they always agree on their decisions about Last and notLast.

Three states, which happen to be transient, involve arbitration or dynamic mutual exclusion (note the “mutex” labels), where the decision of which state must be next is made nondeterministically within Master or Slave. For example, let the S/Start signal arrive first in the initialization mode while Master has just been requested by its client to start data transmission. The decision between sending M/Ack or M/Start is made through an arbitration process in Master. Similarly, when the protocol is in the Master transmit mode (II) and Slave receives a data value that is not last just at the time when its client issues a request for data transmission, Slave must arbitrate to decide whether to send S/Ack or S/Start. The issue of the implementation of arbitration will be discussed later.

Note that the use of symbol SlaveAck is illustrated in the form of a confluent (diamond) structure of state transitions representing the transmission of both S/Start and S/Ack, involving both wires s10 and s01. Such a diamond structure shows an interesting property of this protocol, namely, that it theoretically does not need another arbitration in Slave, which would have been symmetric to the one in the Master when exiting the initialization mode. This is because whether Slave sends an acknowledgement to M/Start or retracts it always produces both symbols S/Start and S/Ack. The latter, transmitted by Slave either in sequence or in parallel, due to the delay-insensitivity of the channel, are assumed to arrive in the protocol machine (and Master) in either order.

3 Protocol Verification

The correctness of the protocol defined in the previous section is seen in terms of the following two main properties that it must satisfy: absence of deadlocks and delay-insensitivity. Both these properties can be verified by constructing a formal model of the communication system. The model of the system consists of the models of Master and Slave and the dual-rail communication channel. In order to adequately capture the behavior of Master and Slave, which may perform some of their actions concurrently, we use the language of Petri nets [13]. This idea is illustrated in Fig. 4a. We constructed two Petri net parts for Master and Slave following the basic protocol in Fig. 3 and inserted two pairs of places, (m01, m10) and (s01, s10), between those parts to represent the two-wire channels. Having a place for each wire in the channel allows modeling a delay between the source of a particular signal wire event and its destination. The former is modeled by a transition that acts as a producer of tokens, e.g., “m01m” indicates sending a signal event on wire m01 by Master. The latter is represented by a consumer transition, e.g., “m01s” stands for receiving a signal event on wire m01 by Slave. The fact that there can be several instances of the same event, activated in different states of the protocol is shown by an index in the transition label, cf. “m01m/1,” “m01m/2,” etc. Additionally, we may use the combined consumer transitions, e.g., “s11m,” to indicate the fact that Master must receive both an event on “s01” and “s10” (SlaveAck symbol).

The Petri net models for Master and Slave parts are built by tracing the protocol in Fig. 3 except that we should adequately model concurrency inside Master and Slave due to interaction with their respective clients. A fragment of the Master model is shown in Fig. 4b. It illustrates the arrival of a request to transmit from the client by the transition labeled “beginm,” which takes Master from state “IdleM” to “wantM.” This transition may fire independently of the arrival of a token in place “s01,” which models the arrival of a S/Start signal from Slave. The fact that there is an arbitration in Master, which must decide whether to send M/Start (i.e., fire transition “m01m/1”) or M/Ack (fire “m10m/1”), is represented by the mutual exclusion construct in the Petri net with a single token in place “meM.”

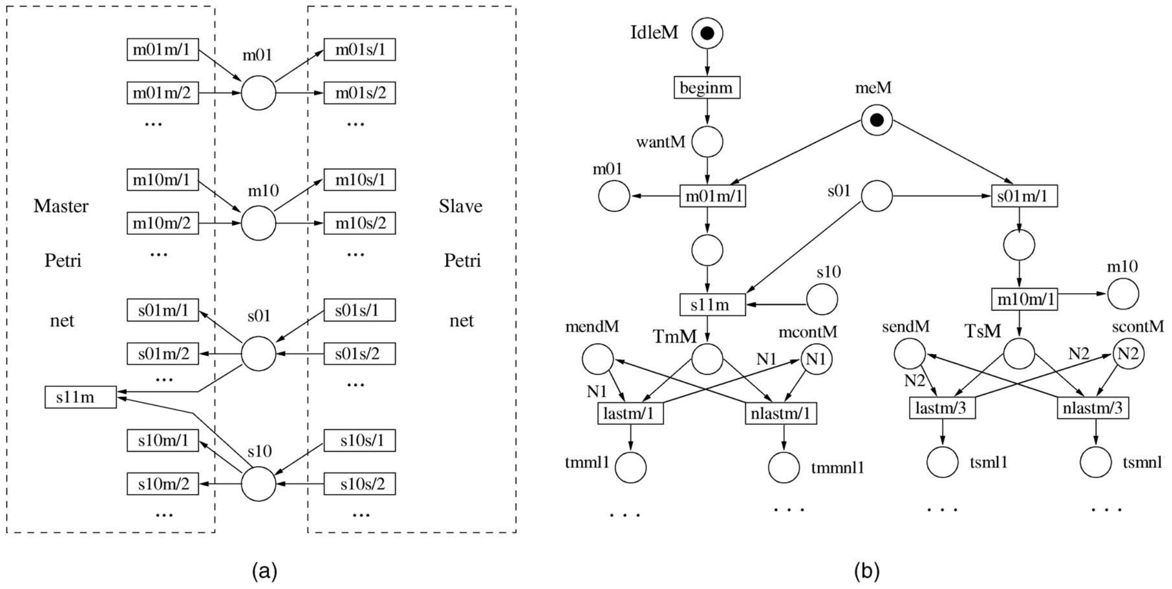

Fig. 4. (a) Petri net modeling of the communication system and (b) fragment of Master subnet.

To model synchronism in counting the length of messages by Master and Slave, we use pairs of places, ("mendM," "mcontM") for the counter which counts bit values transmitted by Master, and ("sendM," "scontM") for counting bit values received by Master. The lengths of transmitted and received messages is "programmed" by N1 and N2, respectively, which set the lengths at N1+1 and N2+1. N1 and N2 also indicate the number of tokens that are initially put into places "mcontM" and "scontM," as well as the weights of the corresponding arcs. For example, in the case of N1, this arrangement of tokens and weights allows transitions "nlastm" (standing for the NotLast data bit value case), which decrement the counter initially set to N1, to fire N1 times before a transition "lastm" (standing for the Last data bit) may fire, which also resets the counter back to N1.

We analyzed the overall Petri net model of the system using the partial order technique based on the construction of a finite prefix of the unfolding (cf. [12]) implemented in the PUNT tool [16]. For this analysis we restricted ourselves with the Petri net model where N1=N2=1, i.e., each message consists only of two bit values. This restriction does not affect the quality of verification because it guarantees that both the Last and NotLast branches of each choice concerned with data transmission and reception are exercised. The tool proved that the net was free from deadlocks.

We verified the delay-insensitivity as follows. If the system was not DI with respect to delays in the wires, any violation would have manifested itself in communication interference [18]. The latter is a transmission event on a channel wire that is not acknowledged by a receiving side and, as a result, another event may occur on the same wire. This condition is easily detected in the Petri net model by means of checking whether the places corresponding to wires m01, m10, s01, and s10 are 1-safe. Indeed, if any such place is not 1-safe this is equivalent to the occurrence of two producing actions on such a place (wire) without at least one consuming action. In its turn, checking whether a given place is not 1-safe is a trivial test in the unfolding prefix; it amounts to finding a pair of mutually concurrent instances of this place. The PUNT tool has proved that the net is 1-safe and, thus, the protocol is DI. The size of the unfolding prefix used for verification was only 84 transition instances.

4 Protocol Agent Design

The protocol agent, one in Master and the other in Slave (cf. Fig. 2b), is divided into three parts, as shown in Fig. 5: main controller, send interface, and receive interface.

The main controller manages interaction with the send and receive interfaces according to the protocol described above. It activates and acknowledges the transmission and reception of data and control symbols, and controls forwarding data from the source system, called Sender, to the output channel, and from the input channel to the destination system, called Receiver. The controller has no direct connection to the data path; this is done entirely via the send and receive interfaces.

The send interface recognizes a request to send data from the source system and informs the main controller about

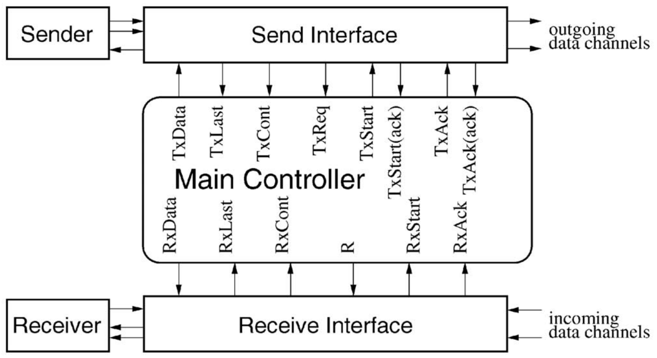

Fig. 5. Overall structure of controller system.

this request using signal TxReq, which starts the initialization procedure. The send interface carries out the appropriate commands of the main controller concerned with issuing data and control symbols. In doing so, it performs phase conversion of data bits arriving from Sender in four-phase RZ dual rail form so they appear in two-phase NRZ dual rail form in the output channel. It also indicates to Sender its readiness to transmit the next bit of data. A counter recording the number of data bits transmitted is incorporated in the send interface (however, alternative ways could be explored, such as the provision of a signal from Sender to indicate the last data bit). The receive interface recognizes the arrival of data from the input channel and carries out commands of the main controller concerned with receiving data and control symbols. In doing so, it converts data bits from two-phase NRZ to four-phase RZ form for Receiver. It also maintains a counter to register the number of bits received.

In order to activate commands of the protocol (see Fig. 3), such as "transmit a Start symbol," the main controller interacts with the interface blocks using handshakes. There are two main types of handshakes. One is called a *push* handshake. It involves performing a transmission action on the output channel. For instance, for "transmit a Start symbol," the main controller uses two handshake signals, request TxStart and acknowledgement TxStart(ack). For another command, e.g., "transmit a Data bit," three handshake signals are used, TxData for request and TxCont and TxLast for acknowledgments. The latter are generated in a mutually exclusive way depending on the value of the transmission bit counter. Namely, if the bit sent to the channel is not the last one, the acknowledgement is sent on TxCont, otherwise on TxLast. A *pull* handshake involves a receiving action on the input channel. For example, if the main controller expects to receive either a Start symbol or an Ack symbol, it uses the handshake consisting of request R and acknowledgements RxStart and RxAck. In another example, let the main controller expect to receive an indicator of a Data bit arriving from the channel (which is passed directly to Receive). Here, the data bit can either be the last in the packet or not depending on the value of the reception counter. In this case, the pull handshake consists of three wires, RxData for request and RxLast and RxCont for acknowledgements.

4.1 The Main Controller

As mentioned above, the main controller follows the protocol. It does not deal with the actual encoding of data and symbols in the input or output data channels. These functions are those of the send and receive interfaces. Such a distribution of functions allows the reuse of the main controller logic in designs where different delay-insensitive encoding is used (cf. M-of-N codes [9]). There are two versions of the controller, one for Master and the other for Slave. The difference between them is only in the initialization part of the protocol. In the sequel, without loss of generality, when talking about the design of the main controller we will refer to that of Master.

In order to proceed to the logic implementation of the main controller, we construct a labeled Petri net (LPN) model of the controller following the main two consistency

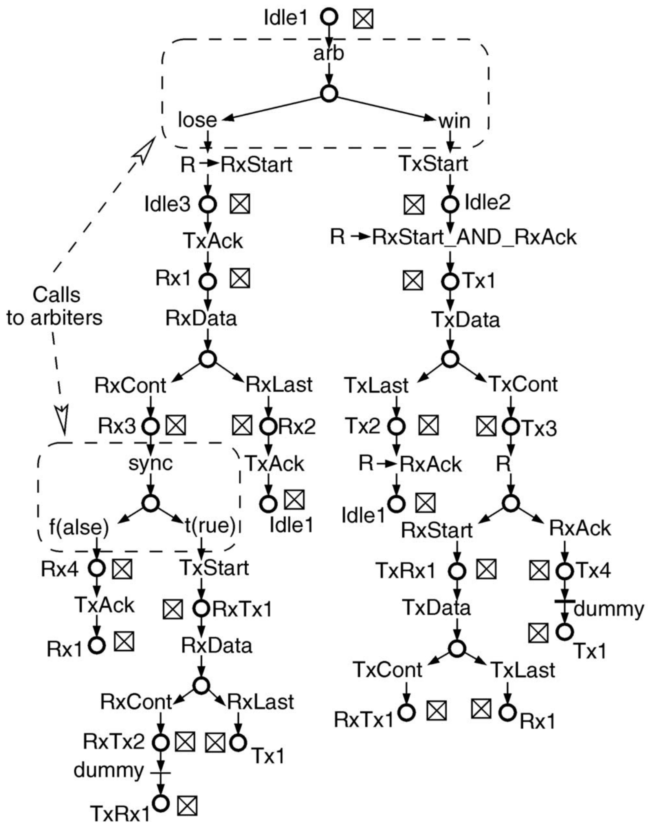

Fig. 6. LPN of Master Control.

requirements. First, it must produce behavior satisfying the protocol defined in Fig. 3 seen from the point of view of Master. Second, it must adequately capture the interaction between the main controller and the interfaces defined by the handshakes in Fig. 5. These requirements are met in the LPN shown in Fig. 6.

An LPN is a Petri net (for formal definitions see, e.g., [13]) whose transitions (events) are associated with the actions or operations of the modeled system, and possibly some silent ("dummy") actions. The places of the LPN correspond to the states that the system may reach in its dynamic behavior. The notion of a marking in an LPN helps to identify the current state of the system by means of tokens in the places. The LPN model can be used as a behavioral specification of the main controller, from which we can derive a structural (logic) implementation using the synthesis technique of direct mapping, as will be described in Section 5. From the point of view of direct mapping, the LPN must be *1-safe*, which means that, in its operation in every reachable marking, every place of the net can never have more than one token. This allows the places of the net to be treated as memory latches in a logic circuit, i.e., each place is either set to logical-0 or to logical-1.

Consider the meaning of the LPN model in Fig. 6. Transitions labeled as "dummy" are internal to the controller as they do not start any actions in the interfaces. The behavior generated by the LPN with these dummy events is equivalent to the behavior defined by the protocol. The dummy events are inserted in order to satisfy the requirement of the direct mapping of LPNs into circuits, where every cycle in the LPN must have at least three transitions [23]. An intuitive explanation of the "three transitions in a cycle" condition comes from the following fact showing the fundamental (but fortunately resolvable) mismatch between the mathematical semantics of the net operation and the physical behavior of the circuit. In a Petri net, the firing of a transition involves a simultaneous act of resetting the marking of the input places to zero and setting the marking of output places to one. In the circuit implementation of the net, where the places are associated with memory latches, the resetting and setting of the latches cannot be performed simultaneously and, hence, some transient states are possible. This requires extra memory to be introduced in extremely short cycles.

The nondummy transitions are labeled as follows: For instance, the $TxStart$ label activates a two-wire push handshake on the send interface side. Its meaning is that of a command to "transmit the M/Start symbol." The $R \rightarrow RxStart$ label corresponds to the activation of a two-wire pull handshake on the receive interface side in order to pull the expected S/Start symbol. The $R \rightarrow RxStart\_AND\_RxAck$ refers to a three-wire pull handshake which pulls the expected SlaveAck symbol. A number of handshakes involve three transitions, e.g., $RxData$, $RxCont$, and $RxLast$, which correspond to pulling the value of the receive data counter. This is modeled by nondeterministic choice, with the decision made outside the controller. Finally, there are two groups of three transitions for the three-wire handshakes with arbitration blocks, which are part of the Main Controller logic but they are implemented outside the logic mapped from the LPN in Fig. 6.

The behavior described by the LPN is cyclic. For convenience, to avoid cluttering of arcs the LPN is shown in Fig. 6 in acyclic form using the identifying labels for the same places (e.g., Idle1, Rx1, etc.). The top place Idle1 is set to logical one in the initial marking.

4.2 The Send Interface

The send interface consists of a counter, a 4-2 phase converter and a Tx-adaptor. Its link with Sender is defined by a standard dual-rail Return-to-Zero (RZ) signaling scheme, with two data bit wires coming out of Sender and an acknowledgement signal going back. The counter controls input signals TxCont and TxLast for the main controller. It is triggered by the TxData signal of the main controller. If the currently transmitted data bit is not the last one, the counter responds with TxCont, otherwise with TxLast. The counter has to be initialized with the appropriate number of data bits in the message. The counter works concurrently with the other components of the send interface. The 4-2 phase converter turns an RZ-signal into a NRZ-signal. That means that every complete pulse on the input is turned into an edge on the output. The converter produces an acknowledgement to the Tx-adaptor for every data bit. The Tx-adaptor is the central component of the send interface. Here, the data signals from Sender are received and acknowledged and the TxReq signal is produced. It also synchronizes control signals from the main controller and data signals from Sender to generate signals to the data channels. It provides an additional handshake between the 4-2 phase converter and the control signals of the main controller.

4.3 The Receive Interface

Similar to the send interface, the structure of the receive interface is derived from the definition of the interfaces between data channels, the main controller and Receiver, which uses a dual-rail RZ scheme. It consists of a counter, a Rx-adaptor, and a 2-4 phase converter. In the initialization phase, any signal which is received and converted into a four-phase signal is forwarded either to the RxStart input or to the RxAck input of the main controller. This process is also valid for the simplex transmission mode. When receiving data either in the duplex or simplex reception mode the received data bits are forwarded to Receiver and the counter gets incremented. The reception of data is acknowledged by the Rx-adaptor. The counter of the receive interface is triggered by the Rx-adaptor. It responds to the main controller with either RxLast or RxCont, depending on whether the currently received data bit is the last one or not. The 2-4 phase converter generates a full pulse (RZ-signal) on the same rail on the output for every edge received on the corresponding input rail. The pulse generation is synchronized with the handshake with the Rx-adaptor.

5 Implementation of the Communication System

This section describes the actual implementation of the communication controller and the send and receive interfaces. Two versions of the controller implementation have been designed and compared. The main version considered in this paper is based on the direct mapping of the LPN into logic (it is called the *place-to-latch* implementation). The other version was obtained by logic synthesis from a Signal Transition Graph refinement of the LPN model, using Petrify [4] (it is called the *minimization* version). The system designs were entered, at the gate level, into the Cadence toolkit and simulated using the AMS CMOS 0.6μ process data.

5.1 Implementation by Place-to-Latch Mapping

A detailed description of the place-to-latch mapping approach can be found in [23], [20], [3]. The direct mapping method is currently being automated [17]. The roots of this approach lie in the seminal works of David [6] and Hollaar [10], who advocated using a modular way (cf. one-hot state encoding) to derive implementations for asynchronous state machines, thus avoiding computationally hard logic synthesis of hazard-free circuits. Today, asynchronous control logic of reasonable size can, in principle, be synthesized using economic state encoding and logic minimization used in the techniques and tools based on STGs [4] (Petrify) and Burst-Mode FSM [14], [8] (Minimalist). They have proved capable of producing robust and compact circuits. However, the size in terms of gates and latches is no longer the predominant factor of control logic designs, where speed, power consumption, and testability can be far more important.

Automatic logic synthesis often generates logic whose depth and, hence, latency are unsatisfactory, and requires human intervention [11]. Direct mapping, whose complexity is linear to the size of the behavioral specification, may be more advantageous in productivity and very competitive in quality against logic synthesis [17]. It allows the

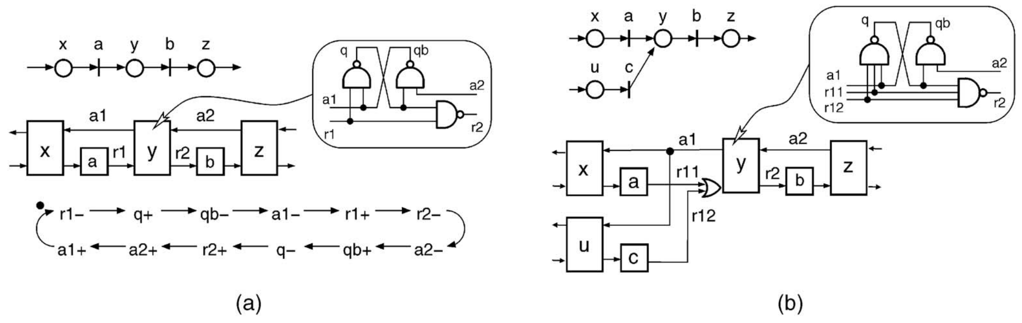
x
a
y
b
z
q
qb
a1
a2
r1
r2
r1− → q+ → qb− → a1− → r1+ → r2−
a1+ ← a2+ ← r2+ ← q− ← qb+ ← a2−
(a)
u
c
r11
r12
(b)

Fig. 7. Direct mapping of LPNs: (a) linear fragment and (b) fragment with merge.

implementation to inherit the topological structure of the specification, thus making the mapped circuit more transparent and easier to verify and test than the result of logic synthesis. The design of the main controller is exactly the right sort of example where the benefits of direct mapping can be demonstrated.

The idea of the place-to-latch mapping is illustrated in Fig. 7, where two typical fragments of the LPN model of the main controller are shown. A linear fragment in Fig. 7a consists of two operations, "a" and "b," activated in series. Each place of the net is mapped to a David cell, labeled according to the name of the place, e.g., "x." The cell is connected to its left and right neighbors via two handshakes. The control of the operations, "a" and "b," is implemented by a simple insertion of the operational logic into the handshakes between the cells, according to the position of the "a" and "b" transitions in the LPN. The logic circuit for David cells used in such a linear fragment is shown on the right. The STG shown at the bottom of Fig. 7a depicts the sequence of events occurring in the David cell when a token arrives in and leaves it. More specifically, after initialization the cell "x" activates operation "a," and request "r1" is generated as logical-0 (active-zero logic is used throughout this paper). This sets the latch built of gates "q" and "qb" in the central cell, "y," to state $\langle 1, 0\rangle$. After this, an acknowledgement is sent to the previous cell, "x," which causes resetting of the "a" operation and propagation of a request to the next operation "b." After that, the latch in the next cell "z" is set. As soon as it happens, an acknowledgement is produced on "a2" in the form of logical-0. This forces the latch "q,qb" in cell "y" to be reset back to its original "empty" state ($\langle q, qb\rangle = \langle 0, 1\rangle$), and the "b" operation is reset making "r2" equal to logical-1. Finally, the latch in cell "z" is reset and "a2" returns to logical-1. The fragment is now in its original state awaiting the arrival of a token in cell "x." This behavior is free from hazards and is totally speed-independent with respect to all gate delays. This is guaranteed by construction and by setting the David cells corresponding to the initially marked places to the states in which signal "q" is set to logical-1 as opposed to the cells associated with empty places, whose initial state is logical-0.

An example of the implementation of a nonlinear LPN fragment ("merge") is shown in Fig. 7b. Here, the token arrives in the place "z" either after operation "a" or operation "c," and then proceeds to operation "b." The corresponding David cell requires more complex gates with three inputs. The reader interested in the logic implementation of a wider range of David cells for LPNs, e.g., involving parallelism, arbitration, etc., can refer to [3]. It should be noted that, in the actual translation of LPNs, not all places need to be associated with a David cell. For example, as will be shown below, the places involved in modeling nondeterministic choice that is made in the environment are not mapped into latches.

The block diagram of the control circuit that is directly mapped from its LPN model in Fig. 6 is shown in Fig. 8. It should be possible to recognize in this diagram the overall structure of the original LPN, which confirms that the direct mapping method preserves the topology of the behavioral model in its implementation. According to this method, most places (those labeled with crossed boxes in Fig. 6) are turned into David cells and most transitions are associated with the actions of the handshakes with the send and receive interfaces. To help reading the block diagram in Fig. 8, an arrow indicating the direction of control flow through David cells are shown in their symbol.

Several key features of the control structure in Fig. 8 are explained in Figs. 9 and 10. The structure of a generic David cell, catering for linear, choice, and merge fragments with several predecessors and successors, is depicted in Fig. 9a. All cells must have reset inputs, which either put a token into a cell or clear it from tokens during the reset phase, according to the initial marking of the LPN (cf. state Idle1 in Fig. 6). David cells can also be built out of generalized C-elements and transistor-level circuits [3]. In this work, we give preference to the use of standard logic gates.

The three-wire handshakes with choice made externally are implemented in the David cell framework as shown in Fig. 9b, using C-elements. Consider, e.g., the box with a three-wire handshake whose signals are named r5, a51, and a52. The RxData request (connected to r5) is generated by the controller while in state Rx1 (cell Rx12). The arrival of an appropriate acknowledgement from the receive interface, either on RxCont (a51) or RxLast (a52), causes firing of the relevant C-element and the token is passed to the corresponding successor cell, either Rx2 or Rx3.

Sometimes, the merge or choice at the LPN can be multiway. This results in high fan-in at the gate level and, hence, slows down the operation. Fig. 9c illustrates how a simple refinement at the LPN level can be performed first and mapping to David cells applied after that, as described in the next section. The example shown in this figure

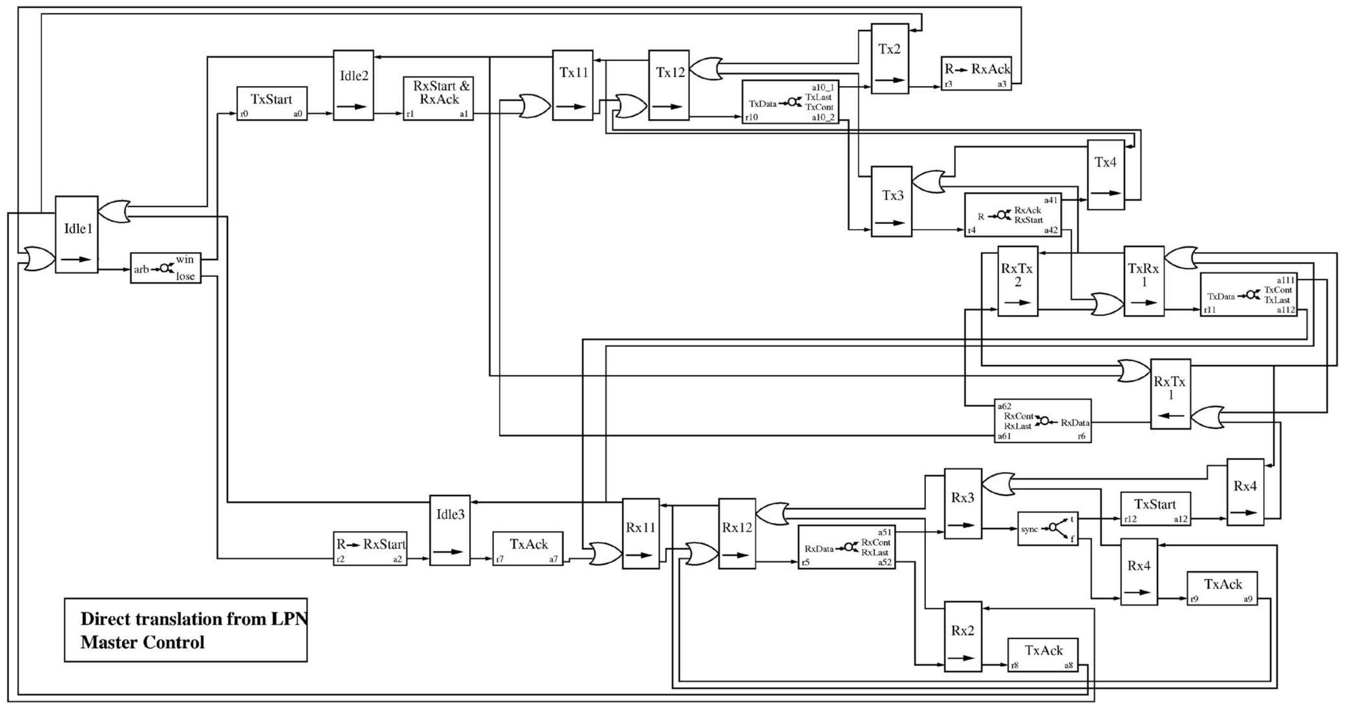

Fig. 8. Block diagram of the main controller.

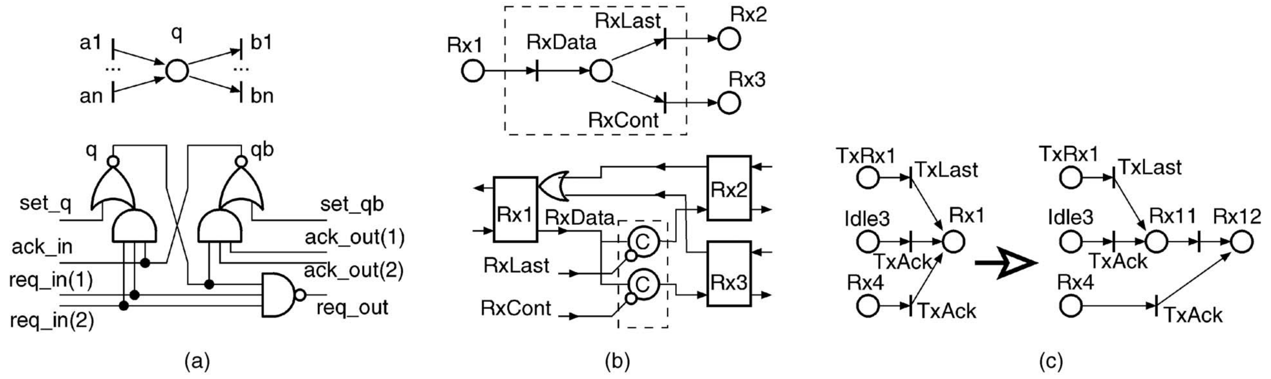

Fig. 9. (a) Generic David cells with reset mechanism, (b) branching structure logic, (c) reducing fanin on places.

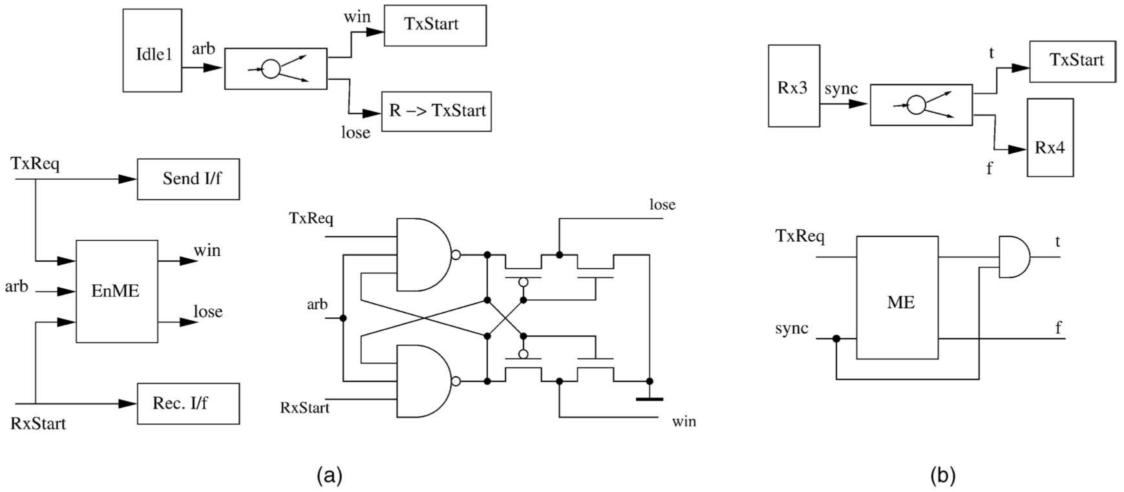

Fig. 10. Arbitration components: (a) initial action arbiter and (b) simplex-to-duplex arbiter.

explains why a single place, Rx1, in Fig. 6 is mapped to two David cells Rx11 and Rx12 in Fig. 8.

The arbitration blocks used in Figs. 6 and 8 are implemented as shown in Fig. 10. There are two points in the LPN specification where arbiters are required. One is the initial action arbitration, where the controller decides whether to enter the Tx or Rx mode from the Idle state. Its design is shown in Fig. 10a. Here, the so-called Mutex with Enabling (EnME) is used. It is activated by signal "arb." The other arbiter, in Fig. 10b, is activated by signal "sync." It makes decision every time new data is received, whether to remain in the simplex (Rx) mode or switch to the duplex (Tx) mode because a request for date transmission (TxReq) has been set.

5.2 Speedup Techniques

In this section, we consider two speed-up techniques taking into account the conditions of the operation of our main controller. The first technique is applied at the LPN level and does not affect the speed-independence of the logic produced by direct mapping. The second one is used at the level of the David cell implementation and makes realistic assumptions about delays in the system.

5.2.1 Place Splitting and Dummy Insertion

As shown above, LPN places can be split in order to avoid occurrence of large fan-ins in David cells. The key paths of the token flow during data transmission involve the places labeled Rx1 and Tx1 in the LPN of Fig. 6. It is easy to observe that the token fan-in in Rx1 and Tx1 is quite large because it involves merging the token flow from three different modes: initialization, simplex, and duplex transfers. The corresponding David cells, Rx1 and Tx1, could therefore have high-fanin gates. A speed-up may be expected if each of these places is split into two places with a lower complexity of the gates in the circuit, as was shown in Fig. 9c. The result of the place splitting is shown in Fig. 11, where new merge places Tx12 and Rx12, and corresponding dummy events are introduced. The reader can compare this LPN with that of Fig. 6.

Another performance issue is concerned with the positioning of dummy transitions in the LPN. Remember that some dummies are inserted into LPN cycles to avoid loops of less than three David Cells because it may otherwise lead to a deadlock in the control logic [23]. On the other hand, it is clear that the presence of a dummy transition between two places implies extra delay between the preceding and the following actions. For example, the presence of a dummy between Tx4 and Tx1 in Fig. 6 means that there is a delay between the processing of the pull ($\mathrm{R} \rightarrow \mathrm{RxAck}$) handshake (before place Tx4) and the push handshake started by TxData (after place Tx1). This delay is on the critical path of the protocol and, thus, slows down the overall communication in the Tx mode. It would be possible to avoid this effect by changing the position of the dummy, e.g., by inserting it into a split of place Tx3, which is between the push and pull handshakes. This would effectively mean having a "masked" delay, which would work in parallel with the communication channel and Slave. Similar transformation can be done with the dummy between RxTx2 and TxRx1 in Fig. 6, which slows down

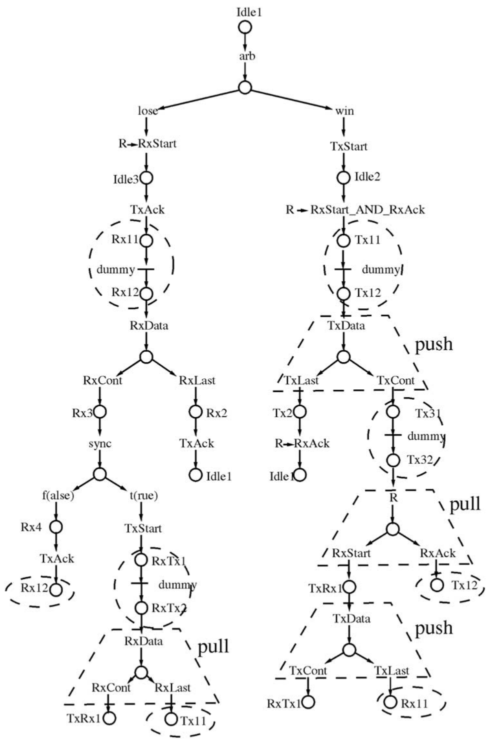

Fig. 11. LPN of Master Control after optimization.

the duplex communication. This dummy is shifted into the split of the RxTx1 place.

5.2.2 David Cells with Early Propagation of Tokens

A number of ordinary David cells, which operate in a speed-independent fashion, can be replaced by David cells with early propagation of tokens. Such fast cells reduce the critical path delay if they are inserted, e.g., between the reception (pull handshake) and transmission (push handshake) operations in the main controller. In ordinary David cells, the activation of the next operation cannot happen until the previous one has finished. The signaling in a fast David cell involves concurrency, where the active phase of the next operation maybe overlapped with the releasing phase of the previous one. This is achieved by exploiting relative timing in the system. Fig. 12 compares an ordinary David cell and a fast one for the case of a linear token flow. The timing diagram on the right shows two timing assumptions (dotted arrows) that are necessary for safe operation of the circuit. The TA1 arrow assumes that the output request, req_out, goes to logical-0 (active state) before the input request, req_in, returns back to logical-1 (stable state). This assumption is true due to the following two reasons. First, the logic path from req_in- to req_in+ (it goes via the previous David cell and the controlled handshake) is much longer than the delay through an inverter and a C-element. Second, at the time of the arrival of a token into the David cell (transition req_in-), signal ack_out is in a stable state (logical-1) as a result of the completion of the previous cycle. This assumption is shown as a TA2 arrow. It holds due to the 1-safeness of the LPN in which every cycle has exactly one token.

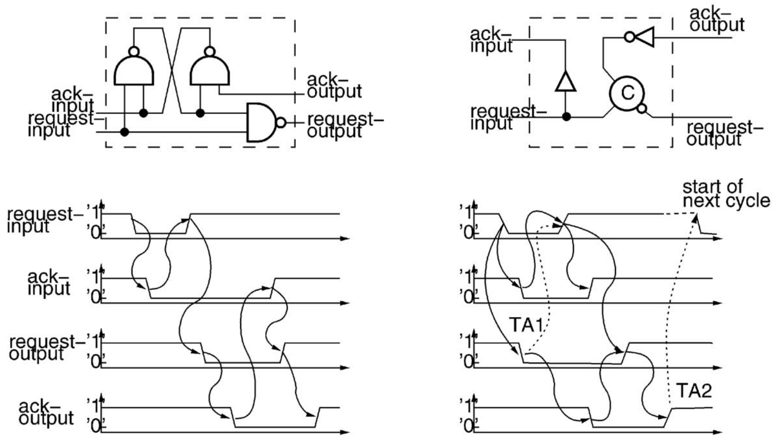

Fig. 12. Comparison between ordinary David cells and cells with fast propagation.

Fig. 13 illustrates, in a schematic way (the paths leading through send and receive interfaces are reduced to simple C-elements), the usage of a fast David cell in a cycle of three cells, similar to those controlling Tx, Rx, and duplex modes. The effect of using fast David cells in such a cycle can be visualised with some sort of a "stretchable" token, which can stretch across two or more David cells. In the totally speed-independent world, such a stretchable token may be risky. Indeed, the place associated with a fast cell may have already propagated the front of the stretchable token (active phase) and started its handshake operation (e.g., pull on R and RxAck), while the previous cell may still keep the back of the token releasing the previous handshake operation (e.g., push on TxData and TxCont). As long as there is an ordinary slow cell in such a loop, which "cuts" the stretchable token and prevents its front to reach its back, the operation is safe. In reality, under the effect of delays, e.g., the delay through channel and Slave, which creates an extra path of signal dependency, such a problem should not happen.

5.3 Implementation of the Send and Receive Interfaces

Send Interface Implementation. The structure of the send interface is shown in Fig. 14. The central component of this interface is the Tx-adaptor shown in Fig. 15. The main functions of this block are: 1) informing the main controller about the arrival of data from Sender (via TxReq), 2) synchronising the process of forwarding 4-phased RZ data signals from Sender to the 4-2-phase-converter with the handshake associated with the TxData commands from the main controller, and 3) forwarding acknowledgements to the main controller when appropriate symbols, Start and Ack have been sent to the channel. For these functions, the Tx-adaptor uses two C-elements, three Set-Unset-blocks (which are C-elements with one of the inputs inverted) and a number of OR gates for merging mutually exclusive 4-phased signals.

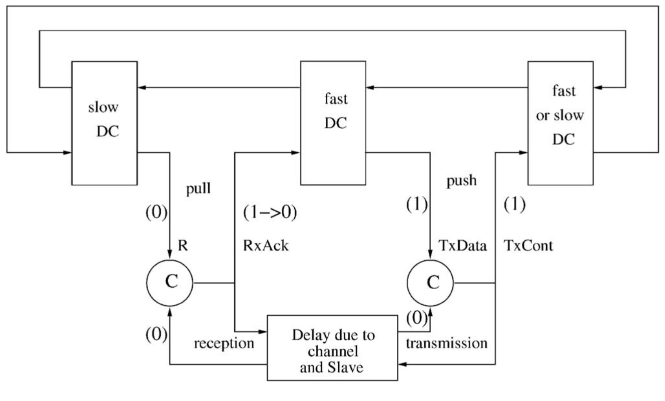

Fig. 13. Illustration of usage of fast David cells.

The TxReq signal, which triggers the transmission process in the main controller, is activated by one of the data inputs, $r0\_in$ and $r1\_in$. Because a request for sending data from Sender may arrive at the same time with the transmission of control symbols by the Tx-adaptor, the set TxReq signal may be released. This can happen due to a delay between recognizing the assertion of the TxReq signal and the reaction of the main controller. To avoid conflict, the release of TxReq is enabled by the TxData signal or by the TxStart signal. Both signals acknowledge the reception of the TxReq signal in the controller. TxStart confirms the acceptance of the first data item in the data packet. The TxData signal indicates the sending of the other items of the data packet. The TxData signal latches the data from the data inputs. Immediately after that, the Tx-adaptor sends an acknowledgement to Sender and the data is removed from the channel. No extra handshake functionality is needed between the counter and Tx-adaptor, which helps these parts to work safely in parallel. The described method is also used for controlling the handshakes for TxAck and TxStart. The appropriate acknowledgement signal, TxAck_Ack and TxStart_Ack, is asserted after the reception of the control signal and is released after the arrival of the acknowledgement from the 4-2-phase-converter.

The counter is implemented using ordinary David cells which form a loop as shown in Fig. 16. The number of David cells in it is equal to the modulus of the counter. The position of the currently activated David cell inside the loop indicates the actual state of the counter. C-elements are placed between the cells in order to stop the token flow until the next trigger pulse arrives. As a response to the

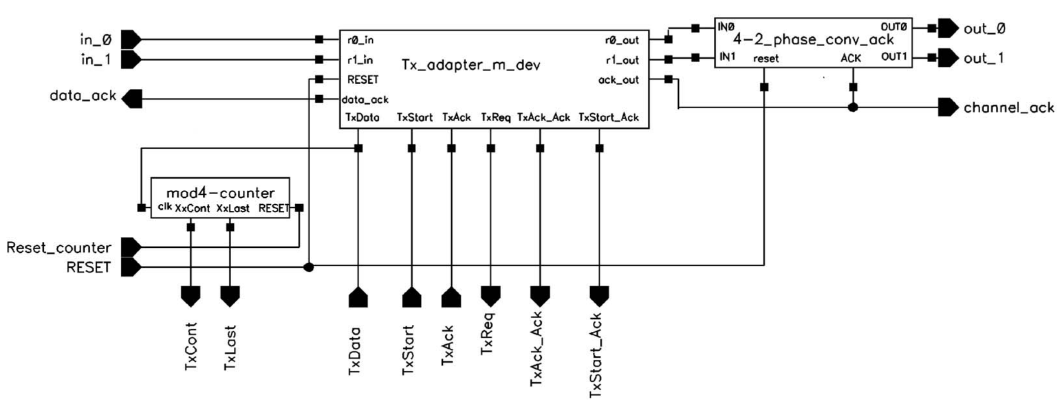

Fig. 14. Send interface logic.

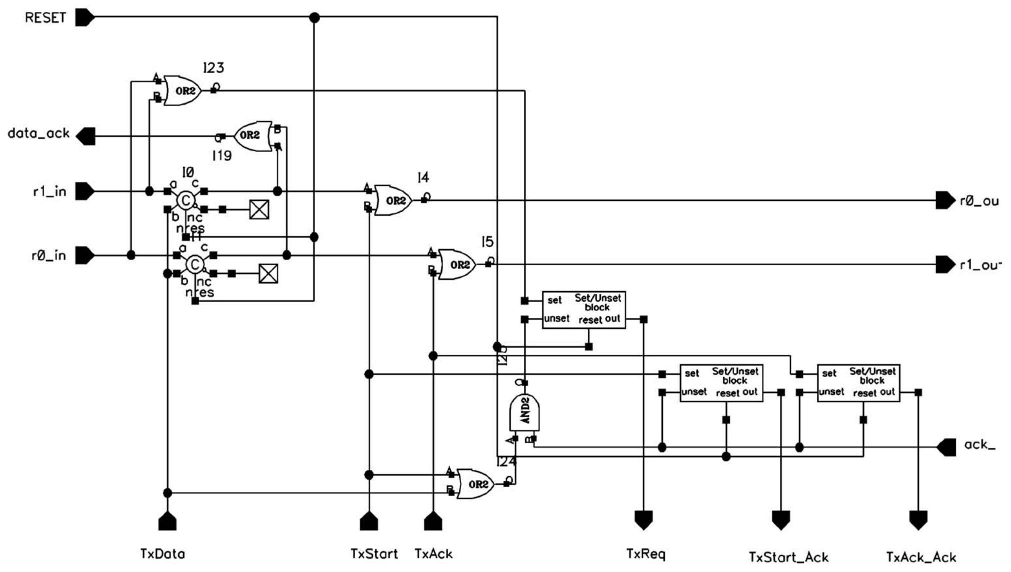

Fig. 15. Tx-Adaptor.

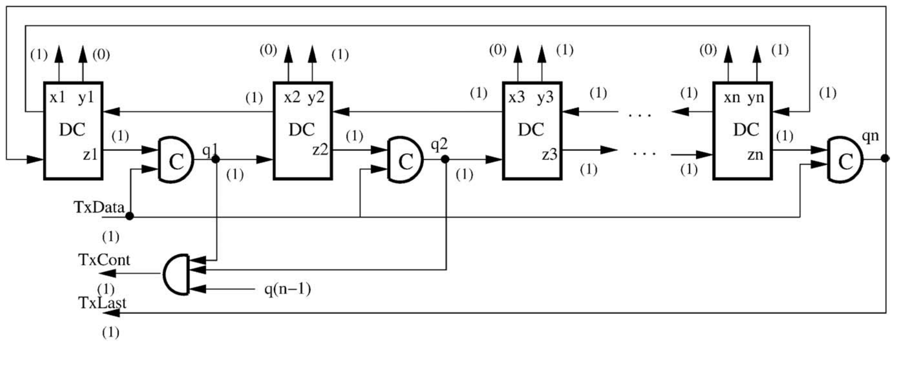

Fig. 16. Mod-n counter built using David Cells.

active level (zero) on TxData, the appropriate output signal (TxCont or TxLast) is activated. This causes a reset of the trigger signal (TxData).

The 4-2-phase-converter for a pair of dual-rail signals is depicted in Fig. 17. It consists of two special elements (toggles) with completion detection, which are assigned to the individual rail channels. For the generation of the completion signal, the individual completion signals of the toggles are combined using an XOR-gate. As the rail channels are mutually exclusive, the XOR-gate produces a completion signal for every transmitted data item.

Receive Interface Implementation. The structure of the receive interface is shown in Fig. 18a. The generation of complete pulses (RZ signaling) for every incoming edge (NRZ signaling) on the individual channel is done using a combination of a D-Latch and an XOR-gate in the 2-4-phase-converter, shown in Fig. 18b. The change of the state of the channel will be directly forwarded to one of the inputs of the XOR-gate. This causes a logical-1 at the output of the XOR-gate. The completion signal (ack_in_0 or ack_in_1) from the Rx-adaptor triggers the adoption of the state of the channel by the D-Latch circuit. The output of the D-Latch is

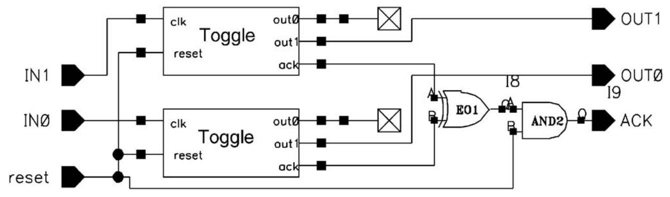

Fig. 17. 4-2 phase converter.

connected to the other input of the XOR-Gate, which enables the latter to complete the 4-phased RZ signaling used between the 2-4-phase-converter and the Rx-adaptor. This circuit does not use any timing constraints. The falling edge of the pulse is generated only after the acknowledgement of the rising edge has arrived from the Rx-adaptor, which in turn makes this acknowledgement only after a "rendezvous" with a request from the main controller (cf. pull handshake) via one of C-elements in the Rx-adaptor, as explained below.

The Rx-adaptor is shown in Fig. 19. It has two groups of C-elements for both signal channels. One group is enabled when the controller is waiting for a control symbol, RxStart or RxAck. The other set of C-elements is enabled if the input signals are interpreted as data. When a control symbol is expected, the R signal of the main controller is kept asserted until one of those signals is received (the effect of a pull handshake). If data symbols are expected, the RxData signal of the main controller is asserted and the data is forwarded to the destination system. The counter of the receive interface is incremented in parallel with this forwarding.

5.4 Implementation by Logic Minimization

The controller built by the direct mapping method has been compared to the implementation obtained from the logic synthesis tool Petrify [4]. For that, the LPN was refined to the level of binary signals that control the handshakes between the main controller and send and receive interfaces. Such a refinement produced an STG with 17 signals, including inputs and outputs of the handshakes and signals generated to and received from two arbitration blocks (described earlier). This STG was then supplied to Petrify for deriving logic equations for the circuit. The initial STG specification had a large number of state coding conflicts (different states having the same binary code). Petrify resolved such conflicts by inserting additional signals. Unfortunately, in models like ours, with most handshake signals having multiple occurrences in the STG, the quality of the automatic resolution of state coding conflicts was poor. An interactive way of resolving state coding conflicts was used, where the designer chose the positions in the STG for inserting new signals. Some manual decomposition was applied at the STG level. This resulted in an STG having

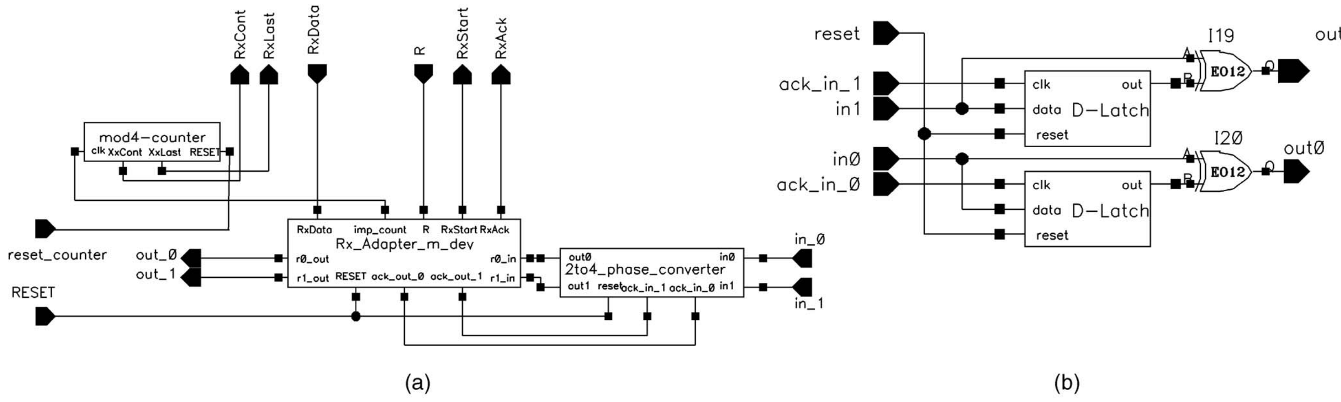

Fig. 18. (a) Receive interface and (b) 2-4 phase converter.

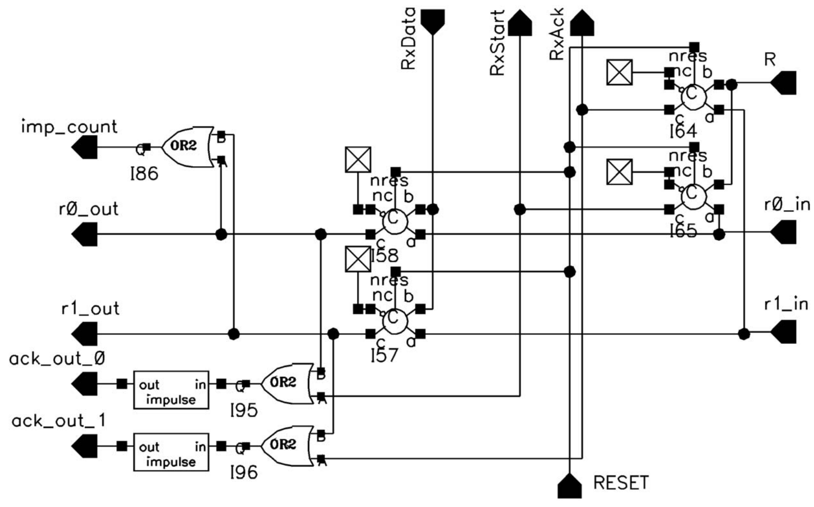

Fig. 19. Rx-Adaptor.

28 signals, for which Petrify produced logic of reasonable complexity. Some circuit-level decomposition was also used, relying on timing assumptions, such as long delays between outputs and inputs due to the path going through the channel and Slave. The analysis of performance in the next section shows that the logic synthesis solution worked significantly slower than the direct mapping solution. Also, given the amount of manual effort spent on guiding Petrify, this clearly demonstrates the superiority of the direct mapping approach in designing this type of controllers. The reader interested in the principles of STG-based synthesis can refer to [4]. An STG-based design of a similar kind of controller is described in [7].

6 Analysis of the Communication System

The functional correctness of the control logic was guaranteed by using direct mapping and logic synthesis techniques [4], [23]. The use of David cells with fast propagation was justified by realistic timing considerations. In order to assess the performance of our designs, i.e., to estimate the contribution of the main controller, send and receive interfaces and channel interconnects toward the critical path delay, as well as to perform comparisons between the place-to-latch circuit and the one synthesized by Petrify, we simulated our circuits using a Spectre analogue simulator in Cadence. We used standard cells for AMS-0.6μ CMOS technology for practically all logic components. C-elements were designed in static logic using transistor meshes [19]. Some cells, such as Mutexes for arbiters, were custom-designed at the transistor level. We did not generate the layout of the design, therefore parasitics within logic blocks have not been extracted. We also decided to minimize the effect of the channel wires to the extreme case by assuming that the composite wire delays in the channel were zero. This assumption puts our duplex system in the least favored position for comparison with a pair of ordinary handshake channels in terms of cycle time, as will be shown below. This helps us to avoid any bias toward our design and should encourage potential users to consider various scenarios of the practical use of this design for on-chip and off-chip interconnects.

The overall functionality of the circuits was checked by extensive simulations in all major data exchange modes (Master transmit, Master receive, and duplex). Switching between the modes has also been simulated. The main objective of these experiments was to measure the cycle time of the communication system during the transfer modes as well as composite delays introduced by key parts of the logic for subsequent analytic estimates. A typical experimental setup is shown in Fig. 20, which depicts the overall system interconnection and the top structure of the Master unit (Slave is similar). After an initial global reset, both Master and Slave were started by separate signals in order to imitate different starting conditions, e.g., to test possible collision of Start symbols and retraction of Slave. In order to emulate the source of data, Sender, a toggle block was used whose function was to alternate between bits equal to 0 and 1.

The analysis of the simulation results consists of three main parts: 1) comparison of the solutions obtained by direct mapping and logic synthesis, 2) comparison of the duplex system to a pair of ordinary handshakes, and 3) estimation of the cycle time based on component delays.

The simulation results required for part 1 are summarized in Table 1. They represent cycle (round-trip) times for simplex (one bit sent in one direction) and duplex (one bit sent in each direction) data transmission. It is clear that the difference in cycle time between simplex and duplex is quite small. The total throughput per wire in both directions lies between 81 and 120 Mbits/s/wire, depending on whether the design is fully speed-independent or uses relative timing. Better throughputs could be achieved if we used M-of-N codes with more effective bits transmitted in parallel (of course, with some overhead in send and receive interfaces for encoding, decoding and completion detection). A figure of 47 MBits/s/wire was reported in [7] for 0.35μ technology, where the circuit was designed by logic synthesis with Petrify. Basically, the performance gain achieved by the direct mapping solution comes from the simplicity of the individual elements of its control logic, which is "thinly" distributed between David cells, forming effectively a one-hot state register. In complex controllers, the gain can be significant where the use of minimization method requires solving a large number of state coding conflicts and where the same control signals occur in the Petri net specification in many places.

In part 2, we have constructed and simulated a dual-rail NRZ handshake channel (with three wires), with the same Sender and Receiver models, in order to compare two parallel unidirectional channels against our duplex system. The cycle time of the ordinary handshake for the same design technology and zero delay in the channel wires has been 3.42ns. It is now possible to make comparison in terms of pin and power efficiency versus performance penalty. The gain in pin efficiency is 1.5 because two bits of data are transmitted in the duplex channel by four wires while the ordinary handshake system uses six wires. The gain in power efficiency is twofold because for two bits there are two transitions in the duplex system versus four transitions in the ordinary handshake system. Thus, the overall gain in pin and power efficiency is about three times. Now, if we take the most conservative version of the duplex system with the cycle time of 12.35ns against 3.42ns, the performance loss would be about 3.6 times. For the version with (very reasonable) relative timing assumptions, the penalty drops down to 2.4. Taken in trade off with pin and power efficiency, this demonstrates the benefits of the use of the duplex system. Moreover, with the use of realistic delays (e.g., off-the-chip) in wires, the advantages of the duplex system would be more pronounced.

In part 3, the critical path of the communication system is estimated by means of the following component-based analysis. The overall communication is controlled by the handshakes from the main controller, i.e., the David cell circuit. These handshakes can be subdivided into five main types of operation sequences, which involve logic in the main controller and in the circuits at lower levels of control. Each type involves a certain path through a number of

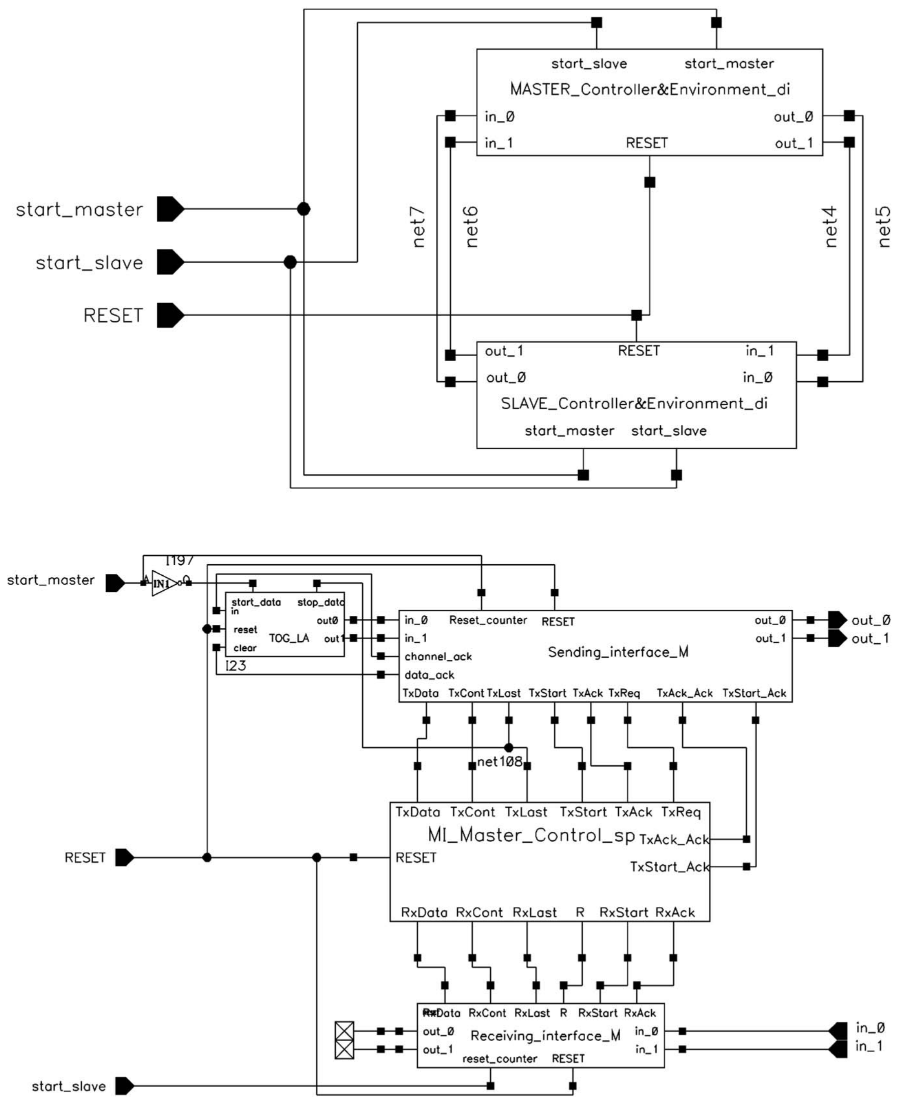

Fig. 20. Experimental setup.

handshakes in the circuit and can be characterized by a certain delay. These types together with their corresponding delays (obtained from simulations) are as follows:

1. TxData-sequence—1.88ns,
2. RxData-sequence—2.70ns,
3. RxStart-RxAck-sequence—1.51ns,
4. TxStart-TxAc-sequence—3.32ns, and
5. dummy-sequence—0.33ns.

TABLE 1
Cycle (Round-Trip) Times Obtained from Simulations

version	simplex mode	duplex mode
place-to-latch: speed-independent	9.99ns	12.35ns
place-to-latch: with relative timing	7.67ns	8.32ns
logic minimisation	12.7ns	16.5ns

For example, the RxData-sequence involves a handshake chain at three levels. It starts at the level of the main controller, which generates a request to the Rx-adaptor level, which subsequently generates another request to the mod8-counter level. The data channel is synchronized at the level of Rx-adaptor. The time delay is the result of adding the time until the Rx-adaptor is ready to receive signals and the time from the arrival of a signal on the data channel until the handshake is complete at the main controller level. Another example, TxStart-TxAck-sequence, involves the following three levels. The controller level is connected in parallel to the Tx-adaptor level and to the 4-2 phase converter. On the way back to the controller level, the 4-2 phase converter is connected in series to the set/unset block in the Tx-adaptor level. The dummy sequence involves only the controller level (two simple gates) and has therefore a very small delay.

The above sequences can be used as "building blocks" for computing various delays in the system when performing simplex or duplex communications. For example, by tracing the sequence of operations in the main controller structure (Fig. 8) of Master and Slave, a duplex critical cycle could be computed out of two TxData-sequences, two

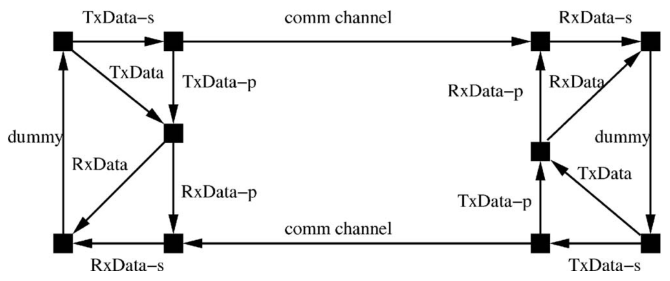

Fig. 21. Segmentation of delays in the system with parallelism (for duplex transactions).

RxData-sequences, two dummy sequences, and two communication channel delays (equal to zero). Adding all these values would give 9.82ns. This would however be an overestimation of the critical path delay because it does not take into account parallelism that exists between switching actions in the system. Fig. 21 shows the segmentation of the delays in the system, where certain parts of the above sequences are performed in series and other parts in parallel with other actions. Let us denote, e.g., the serial part of TxData by TxData-s and parallel by TxData-p; similarly, for RxData. The delays of 1.28ns and 1.98ns are valid for TxData-s and RxData-s, respectively. A more accurate estimation should therefore be as follows:

$$\begin{aligned} t_{duplex} &= max(2t_{TxData-s} + 2t_{comm-chan} + 2t_{RxData-s} + 2t_{dummy}, \\ &\quad t_{TxDATA} + t_{RxData} + t_{dummy}) \\ &= max(2 * 1.28 + 2 * 0 + 2 * 1.98 + 2 * 0.33, \\ &\quad 1.88 + 2.70 + 0.33) = max(7.18, 4.91) = 7.18ns. \end{aligned}$$

This value, taken from the delays of path segments, is not vastly different from the one shown in the simulation table for duplex interaction. Similar estimates can be derived for other modes. Note that the dummies were positioned in the critical path in the above implementation model. Hiding them according to the technique described in Section 5.2.1 would help reduce the cycle time to 6.52ns.

7 Conclusion

The paper presented the design and analysis of a self-timed duplex communication system. The system offers the advantages of power and pin efficiency by using a nontrivial protocol which allows the communicating modules to send acknowledgements of the received data at the same time with transmitting their own data. We presented a formal definition of the protocol using the so-called protocol state machine. The protocol was formally verified using a Petri net unfolding analysis tool. We then derived a logic implementation for the Petri net specification of the protocol controller, and constructed send and receive interfaces between the clients, channel, and the controller. We produced two implementation versions for controller, one using place-to-latch (direct) mapping from the Petri net and the other using logic synthesis from Signal Transition Graphs with the Petrify tool. These versions were entered to the Cadence toolkit and analyzed by SPICE simulation. Analysis of performance showed two important results: 1) the duplex system presents a good trade off between pin and power efficiency gains and performance penalty against a pair of unidirectional handshakes, and 2) the direct mapping method, which fits naturally into the design process based on labeled Petri nets, can give circuit implementations of relatively complex controllers that are faster than those obtained with logic synthesis using Petrify.

This paper showed two main techniques for improving the speed of the controller in place-to-latch version. First, one can transform the structure of the Petri net by place fan-in reduction and dummy repositioning. Second, the basic structure of David cells can be optimised using relative time assumptions available from the knowledge of the environment delays. It is also possible to improve the speed of David cells further by applying transistor-level techniques reported in [3].

Acknowledgments

The authors acknowledge support of EPSRC (GR/L93775/R16754) and the EU-funded Working group on Asynchronous Circuit Design (ACiD-WG).

References

[1] J. Bainbridge, *Asynchronous System-on-Chip Interconnect.* Springer-Verlag, 2002.
[2] L. Benini and G. De Micheli, “Networks on Chips: A New SoC Paradigm,” *Computer,* vol. 35, no. 1, pp. 70–78, Jan. 2002.
[3] A. Bystrov and A. Yakovlev, “Asynchronous Circuit Synthesis by Direct Mapping: Interfacing to Environment,” *Proc. Eighth IEEE Int'l Symp. Asynchronous Circuits and Systems,* pp. 127-136, 2002.
[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic Synthesis of Asynchronous Controllers and Interfaces.* Springer-Verlag, 2002.
[5] A. Danthine, “Protocol Representation with Finite-State Models,” *IEEE Trans. Comm.,* vol. 28, no. 4, pp. 632-643, 1980.
[6] R. David, “Modular Design of Asynchronous Circuits Defined by Graphs,” *IEEE Trans. Computers,* vol. 26, no. 8, pp. 727-737, Aug. 1977.
[7] A. Efthymiou, “The Design of a Low-Power Asynchronous Communication System,” *Proc. Ninth UK Asynchronous Forum,* pp. 1-5, Dec. 2000.
[8] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, B. Lin, and L. Plana, “Minimalist: An Environment for the Synthesis, Verification, and Testability of Burst-Mode Asynchronous Machines,” Technical Report TR CUCS-020-99, Columbia Univ., New York, July 1999.
[9] S.B. Furber, A. Efthymiou, and M. Singh, “A Power-Efficient Duplex Communication System,” *Proc. Int'l Workshop Asynchronous Interfaces (AINT'2000),* 2000.
[10] L.A. Hollaar, “Direct Implementation of Asynchronous Control Units,” *IEEE Trans. Computers,* vol. 31, no. 12, pp. 1133-1141, Dec. 1982.
[11] A. Madalinski, A. Bystrov, V. Khomenko, and A. Yakovlev, “Visualization and Resolution of Coding Conflicts in Asynchronous Circuit Design,” *Proc. Design, Automation and Test in Europe Conf.,* pp. 926-931, Mar. 2003.
[12] K.L. McMillan, *Symbolic Model Checking.* Kluwer AP, 1993.
[13] T. Murata, “Petri Nets: Properties, Analysis and Applications,” *Proc. IEEE,* vol. 77, no. 4, pp. 541-580, 1989.
[14] S.M. Nowick, “Automatic Synthesis of Burst-Mode Asynchronous Controllers,” PhD thesis, Dept. of Computer Science, Stanford Univ., 1993.
[15] L.Y. Rosenblum and A.V. Yakovlev, “Signal Graphs: From Self-Timed to Timed Ones,” *Proc. Int'l Workshop Timed Petri Nets,* pp. 199-207, July 1985.
[16] A. Semenov, “Verification and Synthesis of Asynchronous Control Circuits Using Petri Net Unfoldings,” PhD thesis, Univ. of Newcastle upon Tyne, July 1997.

[17] D. Sokolov, A. Bystrov, and A. Yakovlev, "STG Optimisation in the Direct Mapping of Asynchronous Circuits," *Proc. Design, Automation and Test in Europe,* pp. 932-937, Mar. 2003.

[18] J.-T. Udding, "Classification and Composition of Delay-Insensitive Circuits," PhD thesis, Eindhoven Univ. of Technology, 1984.

[19] K. vanBerkel, "Beware the Isochronic Fork," *Integration, the VLSI J.,* vol. 13, no. 2, pp. 103-128, June 1992.

[20] V.I. Varshavsky and V.B. Marakhovsky, "Asynchronous Control Device Design by Net Model Behavior Simulation," *Application and Theory of Petri Nets 1996,* J. Billington and W. Reisig, eds., June 1996.

[21] T. Verhoeff, "Delay-Insensitive Codes—An Overview," *Distributed Computing,* vol. 3, no. 1, pp. 1-8, 1988.

[22] A. Yakovlev, V. Varshavsky, V. Marakhovsky, and A. Semenov, "Designing an Asynchronous Pipeline Token Ring Interface," *Proc. Int'l Conf. Asynchronous Design Methodologies,* pp. 32-41, May 1995.

[23] A.V. Yakovlev and A.M. Koelmans, "Petri Nets and Digital Hardware Design," *Lectures on Petri Nets II: Applications. Advances in Petri Nets,* W. Reisig and G. Rozenberg, eds., 1998.

Alex Yakovlev received the MSc and PhD degrees in computing science from the Electrotechnical University of St. Petersburg, Russia, where he worked in the area of asynchronous and concurrent systems since 1980 and, in the period between 1982 and 1990, held positions of assistant and associate professor at the Computing Science Department. He is a professor of computer systems design in the School of Electrical, Electronic and Computer Engineering at the University of Newcastle. He first visited Newcastle in 1984/1985 for research in VLSI and design automation. After coming back to Britain in 1990, he worked for one year at the Polytechnic of Wales (now University of Glamorgan). Since 1991, he has been at the Newcastle University, where he is now heading the Microelectronic Systems Design Research Group. His current interests and publications are in the field of the modeling and design of asynchronous, concurrent, real-time, and dependable systems. He has chaired program committees of several international conferences and is currently a chairman of the streering committee of the Conference on Application of Concurrency to System Design. He is a member of the IEEE. In 2002, he was among the finalists of the Descartes Prize, awarded to best scientific teams in Europe.

Steve Furber received the BA degree in mathematics in 1974 and the PhD degree in aerodynamics in 1980 from the University of Cambridge, England. He is the ICL Professor of computer engineering in the Department of Computer Science at the University of Manchester. From 1980 to 1990, he worked in the hardware development group within the R&D Department at Acorn Computers Ltd., and was a principal designer of the BBC Microcomputer and the ARM 32-bit RISC microprocessor, both of which earned Acorn Computers a Queen's Award for Technology. Since moving to the University of Manchester in 1990, he has established the AMULET research group which has interests in asynchronous logic design and power-efficient computing. He is a fellow of the Royal Society, the Royal Academy of Engineering, and the British Computer Society, a Chartered Engineer, and a senior member of the IEEE. In 2003, he was awarded a Royal Academy of Engineering Silver Medal for "an outstanding and demonstrated personal contribution to British engineering, which has led to market exploitation."

René Krenz received the MSc degree from Reading University, U.K., and the Dipl-Ing. (FH) degree from FHTW-Berlin, Germany, in 2001. He is currently a PhD student in his third year in the Department of Microelectronics and Information Technology (IMIT) at KTH, Royal Institute of Technology, in Stockholm. His research interests include formal verification, logic lynthesis and optimization, VLSI related graph algorithms, and asynchronous circuit design. He is a member of the IEEE.

Alexandre Bystrov, from 1980-1986, studied electronic engineering at St. Petersburg State Electrical Engineering University, Russian Federation. After graduation and until 1995, he was working as a research associate in the Department of Radio Systems of the same university, studying online and offline testing methods. From 1995-1998, he was doing research on the optimal testing of multilevel logic circuits in Napier University of Edinburgh and received the PhD degree in electrical engineering from the same university. He is a lecturer in the School of Electrical, Electronic and Computer Engineering, University of Newcastle upon Tyne. Since then, he has been working in the Asynchronous Systems Laboratory at the University of Newcastle upon Tyne. During the last few years, his research interests have been the modeling, visualization, and design of asynchronous systems, arbitration, design, and testing of low-latency asynchronous circuits and security systems design. He is a member of the IEEE.