School of Electrical, Electronic & Computer Engineering

UNIVERSITY OF
NEWCASTLE UPON TYNE

Asynchronous Checker designs for monitoring Handshake Interfaces

D. Koppad, D. Shang, A. Bystrov and A. Yakovlev

Technical Report Series

NCL-EECE-MSD-TR-2005-104

March 2005

Contact:
Deepali.Koppad@ncl.ac.uk
Delong.Shang@ncl.ac.uk
A.Bystrov@ncl.ac.uk
Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grant GR/S12036

NCL-EECE-MSD-TR-2005-104

School of Electrical, Electronic & Computer Engineering,
Merz Court,
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU, UK

`http://async.org.uk/`

Asynchronous Checker designs for monitoring Handshake Interfaces

D. Koppad, D. Shang, A. Bystrov and A. Yakovlev

March 2005

Abstract

Checker designs for on-line testing of asynchronous handshake interfaces are proposed here. The checker monitors the interface signals that follow a protocol. The checker produces a code word at its output when the interface signals abide to the protocol, where as, when the protocol is violated, a non-code word is generated at the output. Checkers are designed to directly implement sets of forbidden transitions, otherwise known as refusals. A "busy" approach is used to design the checker. In this approach, self-test of the checker is performed during the normal operation where the output signals are constantly switching.

1 Introduction

With rapid advances in technology, billions of transistors can be integrated into a single die. This is achieved by shrinking feature sizes, often at the expense of parameter variability and overall design predictability. The International Technology Roadmap for Semiconductors [1] states that in spite of concerns expressed in its earlier version, the Moore's law is still in place. As a result, designers face challenges of creating very complex dependable systems on a single chip, specifically aimed at fabrication technologies having increased variability and soft-error [2] rate. An approach to alleviate the effect of parameter variations on timing closure is a Globally Asynchronous Locally Synchronous System (GALS) [3], which uses asynchronous and, possibly self-timed interfaces between blocks. Compared to synchronous circuits designed and fabricated today, asynchronous circuits are fundamentally different. This difference gives asynchronous circuits inherent properties that can be (and have been) exploited to gain advantages as no clock skew, low power consumption [11], security features, etc. So far several companies including Sun, Philips and Intel are either investigating the use of asynchronous techniques in their products or are already using them. However, asynchronous approach is not yet well-established and widely-used design methodology. The main reasons for this are: 1) lack of CAD tool to support asynchronous circuit design; and 2) insufficient development of testing methodology [18].

Here we propose on-line testing method for asynchronous circuits and implement checkers to monitor simple handshakes. A typical self-checking circuit includes a functional circuit and a checker circuit. The checker circuit, monitors for code words at the output of the functional circuit. If a non-code is detected, the checker indicates this (fault secure). Also if any faults occur within the checker, then this is also indicated (self-testing).

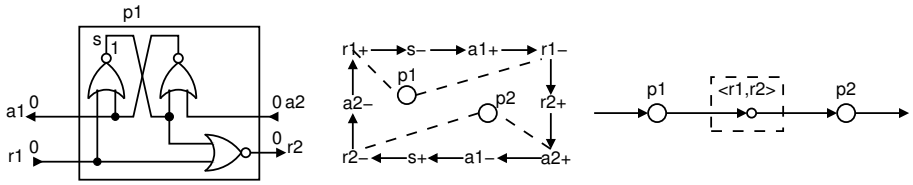
In this report, we apply traditional on-line testing methods to asynchronous circuits and analyse the specifics of such an application. The myth of 'high testability' of self-timed circuits is discussed. Several checkers for handshake interfaces are presented.

2 Background

Explicit logic synthesis [13] and direct mapping [25] are two techniques for synthesizing asynchronous circuits.

Logic synthesis method uses Signal Transition graphs or STGs, as system specification. This method derives logic equations for the output signals using the next state functions. These functions are obtained from STGs, by exploring all possible combinations of firing orders. This may lead to a state space which is larger than the original STG. The state space is represented as a *State graph* (SG). A SG is a binary encoded reachability graph of the Petri Net. Then the theory of regions [14] is used to obtain the logic equations for the output signals. The tool, PETRIFY, uses this synthesis technique.

Petri nets are used as system specification for Direct mapping technique. In this technique, the initially closed model (model that captures both device and environment), is converted into an open model. For this, two transformations, namely, *environment tracking* and *output exposition* are done. These transformations give rise to an open model, consisting of *tracker* and *bouncer*. The tracker precomputes the output signals (in parallel to the environment). The bouncer, uses these signals and generates the output as soon as it receives the trigger signals from the environment. The tracker and bouncer are connected to each other by means of read-arcs, which are equivalent to self-loops. These are non-consuming arcs, with no arrow heads at both ends and which control enabling of transitions. The tracker (Petri Net places) is mapped into David cells [26] and the bouncer to latches.



(a) Circuit (b) STG (c) Model

Figure 1: David cell

Structure of a David cell (DC) is shown in Figure 1 (a). DCs can be coupled using four-phase handshake interfaces, so that the interface <*r2,a2*> of the previous stage can be connected to interface <*r1,a1*> of next stage. It operates under the protocol defined as a STG in Figure 1 (b). Places *p1* and *p2* can be used to model places of PN shown in Figure 1 (c). The dotted rectangle depicts the transition between *p1* and *p2*, containing an internal place where token disappears for time $\tau_{r1-\rightarrow r2+}$. This corresponds to one gate delay and is considered as negligible.

Both the above synthesis methods are used to derive checker designs. Comparison of the two methods in terms of area, latency and testability is given in the section 6.

3 Testability of self-timed circuits

Early studies on this subject [8, 9, 10], discovered interesting testability features of asynchronous circuits with complete acknowledgement property otherwise known as delay-insensitive and speed-independent (SI) [11]. These circuits tend to halt in presence of stuck-at faults on interconnect wires (s-a faults at gate outputs). This formed a foundation to the myth of self-timed circuits being totally self checking. There are several constraints formulated in [8], which need to be satisfied to make a SI circuit TSC (which are often overlooked). These constraints require a s-a- fault to be constant, to be positioned at a logic gate output, and the circuit is required to be deterministic and hazard-free (state graph being semi-modular lattice). The following two examples show how restrictive these constraints are.

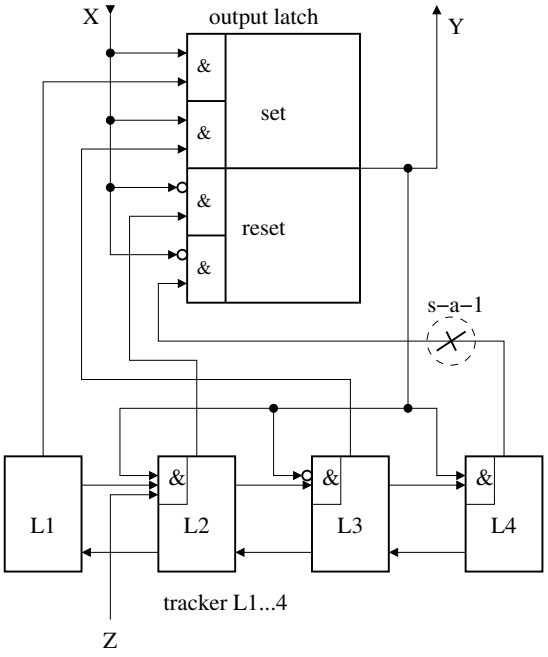


Figure 2: Low-latency circuit

The first example (Figure 2), is a 'tracker' comprising of five self-timed latches, L1-L4 controlling the 'output latch'. X is the primary input and Y is the primary output. This is a simplistic example of low-latency SI design approach described in [12]. Trackers are capable of capturing very complex behaviours, as they directly implement a Petri Net model, moreover, each element of the tracker implements a single place of the model. In our example a single token ('on' state) is shifted through the tracker left to right (one-hot encoding). The feedback from the output latch slows down the request-acknowledgement handshakes between the elements of the tracker, making them to wait for the corresponding output. Input Z us used by the environment to delay the corresponding transition (first Y-) of the protocol. If a s-a-1 fault is introduced at the location indicated by the cross the following will happen. L1 enables Y+ and it switches as soon as X+ takes place; this is normal. Then the environment does X- and, for example, delays Z+. The circuit is expected to keep Y=1 until Z+, but the fault enables Y- and creates the race between Z+ and Y-. As as result, L2 may go meta-stable or not switch at all, and the environment may get a hazard in its protocol checking circuitry. So, in this example a fully deterministic and hazard free SI circuit produced hazards

under a s-a fault at interconnect, which may result in intermittent errors.

In the second example, we look at the behaviour of the DC in presence of a s-a fault at the gate input. If the fault is introduced, as shown in Figure 3, then the circuit stops being SI and may produce a hazard by doing a1+ early as a response to a2- rather than r1-. On the other hand, the circuit behaviour under a fault can be an over-set w.r.t the correct behaviour. One needs to exercise various delays in the signal sources and gates in order to detect the behaviour which has been 'added' to the normal behaviour. This is often difficult, impractical or impossible due to delays being fixed at the moment of circuit fabrication. In the current example, in order to guarantee activation of the fault it is necessary to incorporate a logic gate into the r1-a1 handshake interface and to control it accordingly to provide $\tau_{r2-\rightarrow a2-+} < \tau_{a1-\rightarrow r1+}$. So, the timing closure achieved by using handshake signalling and clock elimination becomes a major obstruction to testability.

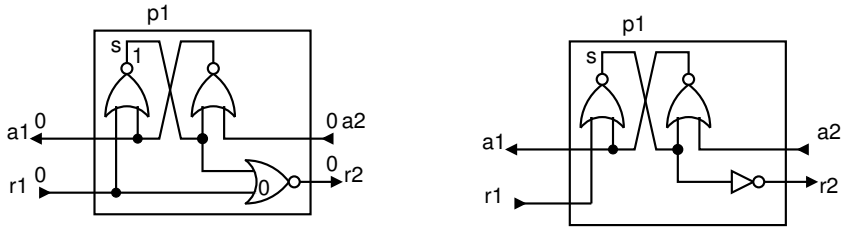


Figure 3: Stuck-at fault in DC

Thus, two fundamental problems of asynchronous circuit testing have been outlined: hazards under a fault and reduced fault coverage. Hazards in synchronous systems are dealt with by introducing a time delay sufficient for the hazards to resolve itself into a stable state. This delay is defined by the clock. There is no clock in asynchronous circuits and even a minute hazard may upset a latch. This can be dealt with by introducing delay elements in the checker structures. For the second problem we currently have not solution apart from using known structures with small number of untestable faults.

4 On-line testing infrastructure

In this section we consider application of a traditional double redundancy approach to asynchronous interfaces. An attempt to solve 'synchronisation' problem leads us to a new model and new checker solutions.

Asynchronous circuits are based upon 'closed' models. If an asynchronous block is duplicated with the purpose of on-line testing, then it needs to be synchronised with the whole system. A direct solution to this problem is shown in Figure 4. Two asynchronous blocks, Block 1 and Block 2, take the same inputs from the environment and their outputs are synchronised pairwise, by means of Muller C-elements.

Synchronisation of the checker with the functional system presents a difficult problem, as due to timing flexibility it may be difficult to convert the outputs into dual-rail code. An example of the synchronisation 'wrapper' for a standard dual-rail TSC checker is shown in Figure 6. STG of Block 1 and 2 is shown in Figure 5, where *a* is an input from the environment and *x, y* are outputs of the blocks. The outputs are synchronised pairwise using C-elements. The blocks produce exactly one switching event on each output wire in each cycle of operation. The functional blocks are considered to be slower than the checker hence their outputs are not acknowledged by the wrapper.

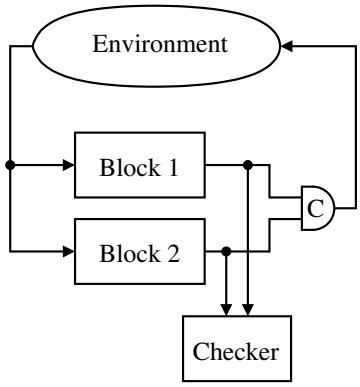


Figure 4: Worse case performance

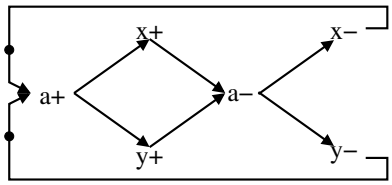


Figure 5: STG of an example

The synchronisation problem w.r.t the synchronic distance [15] is solved by using FIFOs. The outputs are compared by a standard TSC checker. This checker produces a code word at its output if no fault is detected. Whereas in presence of a fault, a non-code is generated on 'Data checker output'. The use of such a checker makes sense only if data types associated with the output transitions are complex. If they are simple 'tokens' (present - not present value), then their comparison is trivial and redundant. The standard dual-rail checker is not needed. In this case, it is important to check whether the tokens were placed in the FIFOs correctly or not. This is the duty of the protocol checkers. For example, suppose the FIFO is full. The FIFO indicates this by raising the *ack* signal. At this moment, the FIFO cannot accept the next set of outputs from the block. But if the outputs are generated (due to an error), the protocol at the input of FIFO is violated, which is detected by the protocol checker. The self-communicating fabric registers this violation and generates a non-code word on 'Protocol checker output'.

So in this structure the concerns of data checking and protocol checking are separated. In our circuits, our main focus is on testing of control path only. This is done by monitoring only the protocols, hence the data checkers from the above structure are not needed. Testing of data path was addressed in [7], and an efficient solution based on redundancy codes was developed.

In our solution (Figure 7), we have removed the data checker, leaving in place the protocol checkers only. This is applicable to interface and control circuit testing where data types are trivial. The second copy of the functional block is also removed and its behaviour is incorporated into the protocol checkers.

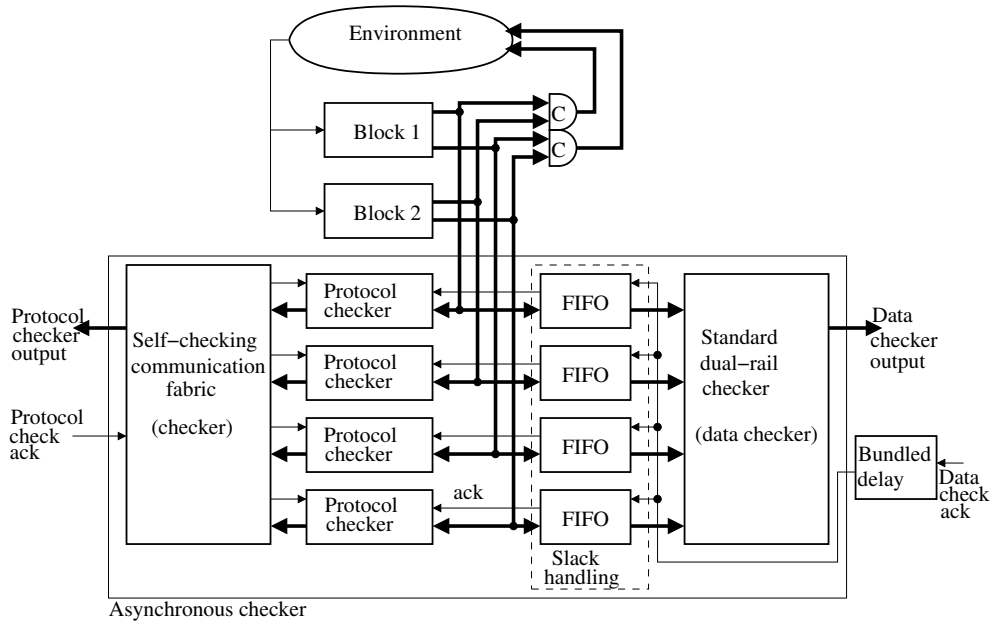


Figure 6: TSC checker wrapper

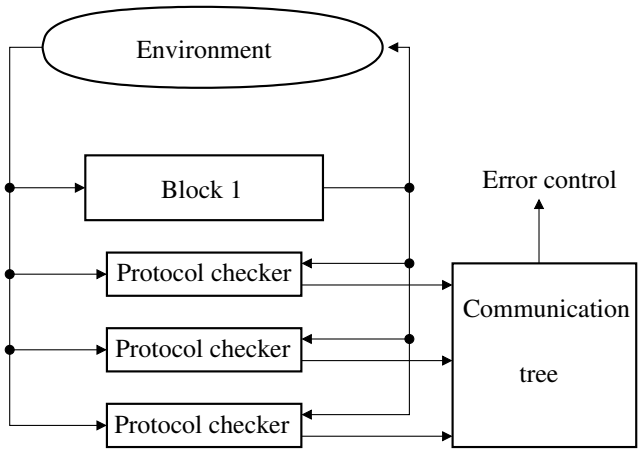


Figure 7: Testing by protocols

5 Checkers for protocol

Throughout this section we use a simple handshake protocol $r+ \rightarrow a+ \rightarrow r- \rightarrow a- \rightarrow r+$ as an object under test. Protocols can be modelled as state graphs [17]. A state graph for simple handshake is shown in Figure 8. For each state one can define a set of enabled transitions and a set of disabled or refused transitions. This is the foundation of the theory of refusals [23]. In our approach, we define a set of refusals for each state of the protocol and construct the model of the checker. For example, in state 00, the enabled set is {r+} and the disabled set is {a+}. The enable transitions lead us to the next valid state, 10, and the disabled transitions to the error state, 01.

At this stage it is impossible to derive a specification of a self-timed circuit because all protocol signals

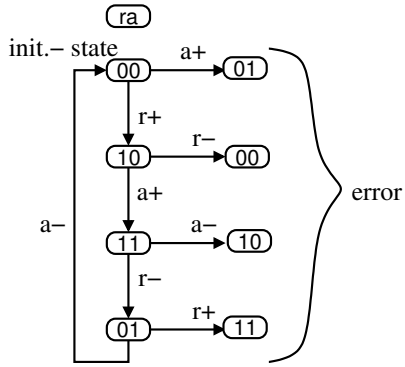


Figure 8: Protocol under test

are inputs w.r.t the checker. It just 'observes' the protocol having no means to acknowledge its own actions. This, however, results in a very important property of non-interference, i.e. the checker does not slow down the functional circuit. It is assumed that the protocol is slow enough to let the checker operate correctly.

Here we propose checker designs based on "busy" signalling. This approach is similar to traditional synchronous checker. It tests itself under normal operation by processing data, which in general case is non-deterministic, i.e. one cannot guarantee that the complete test set is applied in bounded time. Furthermore, such a checker constantly switches its outputs under normal operation (burning power). The checkers based on this approach are self-checking. These checkers are implemented in AMS-0.35μ CMOS technology using Cadence tools. The analogue simulations for these checkers are also presented.

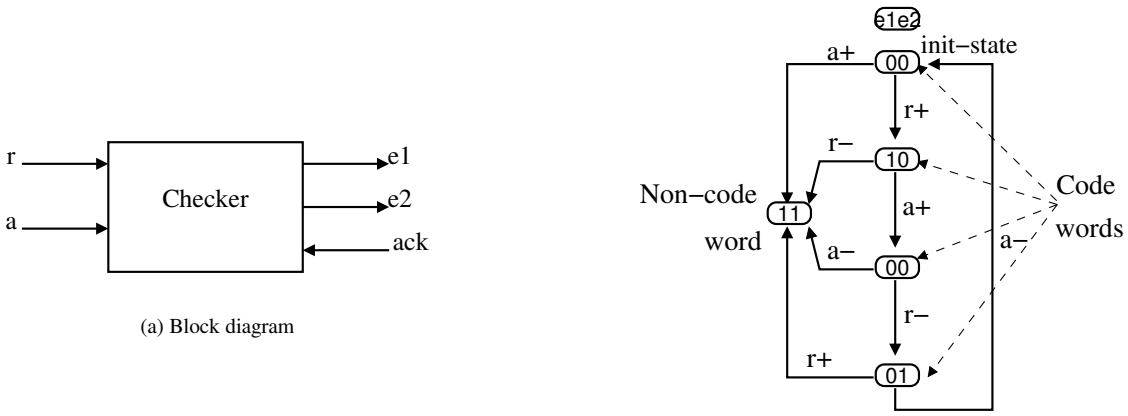
On the contrary, checkers can be designed based on "lazy" approach. In this approach, the checker is self-tested on request only. As such it saves power by not switching its outputs during normal operation, but only under the self-test mode. This gives a bounded time for self-test. During the self-test mode, the operation of the functional circuit is not affected. Checkers based on this approach, sacrifice fault security during the self-test mode. So we can say that these checkers are either fault secure most for the time or self-testing for a short period.

5.1 Checker design

The block diagram of the checker using the busy approach, is shown in Figure 9 (a). The protocol signals (r and a), are the inputs to the checker. The output signals ($e1$ and $e2$) are constantly switching, under the normal operation. The checker produces code words on these signals when the protocol is followed. Any violations by the protocol signals is detected by the checker, and it generates a non-code word on the output signals.

The protocol followed by e1 and e2 is shown in Figure 9 (b). States 00, 10 and 01 belong to the set of code words. Where as state 11 belongs to set of non-code words. For example, when input r+ (which belongs to enabled set of state r=0, a=0) is received, the checker produces the code word 10 (e1=1, e2=0) at its output. However, if input a+ (belongs to refusal set of state r=0, a=0) is received then checker generates the non code word 11 (e1=1, e2=1). Next, if a+ (belongs to enabled set of state r=1, a=0) is received, code word at output of checker is 00 (e1=0, e2=0). If r- (belongs to refusal set of state r=1, a=0) is received then non-code word 11 is generated. Similar sequences for other states can also be derived.

STG of the checker is shown in Figure 10. This design is a direct implementation of the set of enables



(a) Block diagram

(b) Protocol of outputs

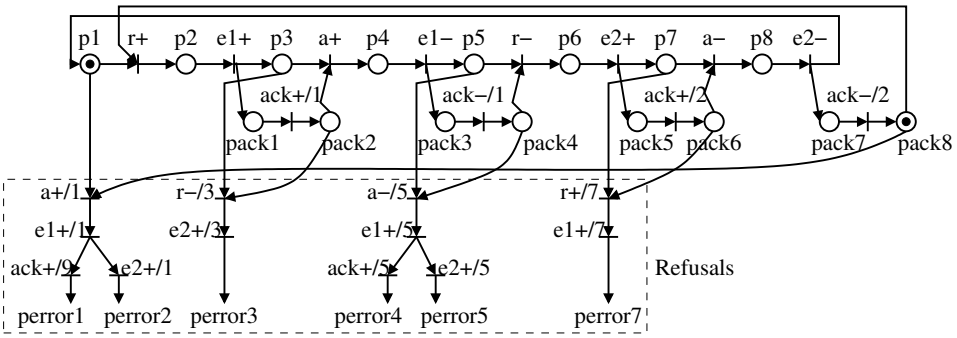
Figure 9: Checker

and set of refusals defined in Figure 8. The checker outputs, follow the protocol shown in Figure 9 (b).

At place p1 (state 00) refusal set is *a+/1* and enable set is *r+*. If *a+/1* is received, *e1+/1* and *e2+/1* fire generating a non-code word at the output. If *r+* is received, token moves ahead to place *p2*, and sets *e1* to 1. This generates a code word, 10, at the output of the checker. An acknowledgement is received on *ack.* The checker is now ready to accept the next input.

Sets of enables, refusals and firing sequences can also be defined for states 10 (place p3), 11 (place p5) and 01 (place p7) similar to those defined for state 00 (place p1).

The checker is synthesised using Petrify [13] which generates logic equations as shown in Figure 10.



[e1] = csc1'a' +csc0a + re2;
[e2] = csc0'(e1a' + csc1) + r'e1;
[csc1] = ackcsc1 + r'ack';
[csc0] = a'(csc0 + r') + r'csc0;

Figure 10: STG of checker

Simulation results Simulation results for the checker are shown in Figure 11. The output environment of the checker, for performing the simulation, is a simple block. This block consists an XOR and AND

gate, outputs of which are ORed together to generate the *ack* signal. In practice, this environment will be more complex.

The waveforms show the expected behaviour of the checker. If the protocol under test is not violated, the checker produces code words on the output signals. When the protocol is violated (*a* is received before *r*), a non-code word (e1=1, e2 =1) is generated.

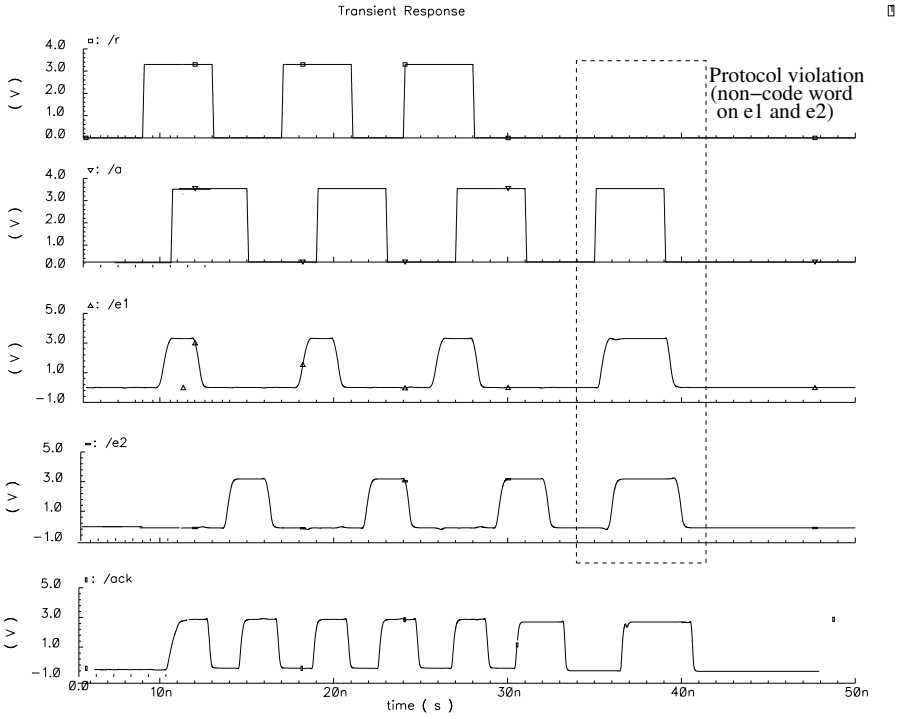


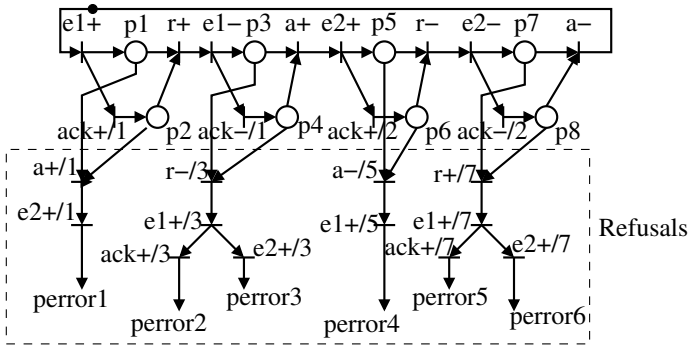
Figure 11: Simulation results of checker

Checker with early propagation The concept of early propagation introduced by [24] can be used to design an optimised checker. In this concept, some signals are precomputed in advance. These precomputed signals and trigger signals from the environment together form the output.

In the previous checker design, it can be observed that, from place p1 (state 00), *e1+* fires irrespective of whether the input is from enables *(r+)* or refusals *(a+/1)* set. Also *ack+* fires immediately after *e1+*. These signals *e1+* and *ack+* can be precomputed before *p1*. Then depending on whether *r+* or *a+/1* arrives, either *e1-* or *e2+/1* can fire. The STG of the checker using precomputation or early propagation concept is shown in Figure 12. The sets of enables and refusals are similar to previous checker. Other signals *e2+* and *e2-* are also precomputed at place p5 and p7.

This checker is also synthesised in Petrify to obtain the logic equations shown at the bottom of the STG.

Analysis of the checker design A circuit is said to be self-checking if it satisfies the properties of fault-secure and self-test [6]. A self-checking circuit should be able to detect errors in the functional block as well as the checker block. Fault secure property of the functional circuit is satisfied using the above checkers.



[e1] = a'r' + a'e2 + rcsc1' + r'e1;
[e2] = csc1e1r' + csc1a + csc1 e2 + ae1;
[csc1] = a'csc1 + a'r + rcsc1;

Figure 12: STG of early propagation checker

In order to evaluate self-testability of the checker, all possible input stuck-at-faults are enumerated. After analyses, we obtain the following results.

For the checker, 53% of the input stuck-at-faults are detected and 47% are not detected. These checkers are obtained using Petrify. It generates equations consisting of complex gates. Complex gates have many inputs and one output. So all input stuck-at faults do not affect the output, resulting in low testability.

A possible solution would be, to design checkers using Direct mapping technique. Circuits obtained by this technique are shown to have higher percentage of self-test [20] with respect to stuck-at faults.

Checker using Direct mapping technique The concept of direct mapping introduced in section 2 is used to derive a new checker design. The Petri Net model of this checker is shown in Figure 13, consisting of a tracker and a bouncer. The tracker precomputes the output signals (transitions shown in brackets in the tracker). The bouncer uses these signals and generates the output as soon as it receives the trigger signals (*r, a* and *ack*) from the environment.

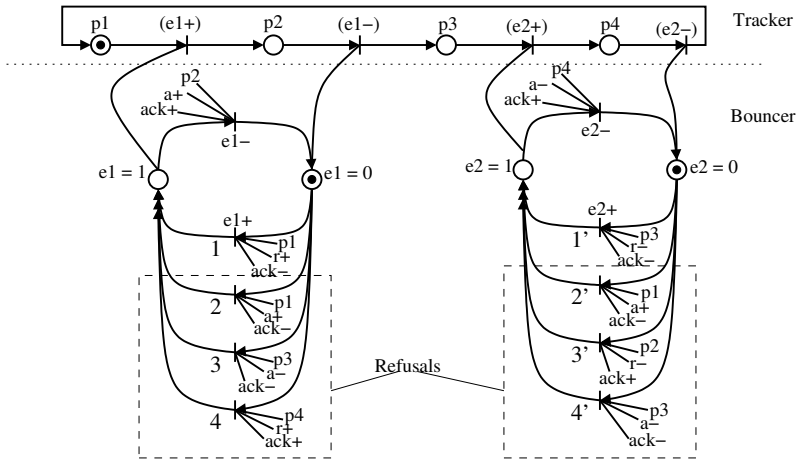


Figure 13: Petri Net model of checker

Sets of enables and refusals are similar to those defined previously. In place *p1*, when input from enables set (*r+*) is received, transition 1 is enabled (setting *e1* to 1). If input from refusals set (*a+*), is received, transition 2 and 2' are enabled. This sets both *e1* and *e2* to 1, generating a non-code word at the output. Similarly other sets of enables and refusals can be defined.

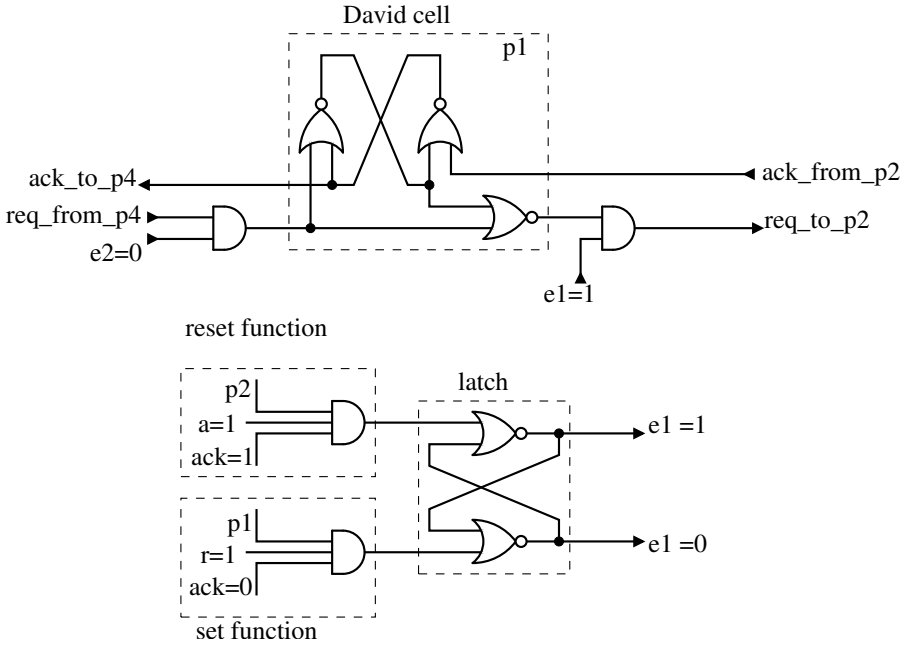


Figure 14: Gate level implementation of checker design

Places *p1* to *p4*, in the tracker are mapped to DC and the elementary cycles (e1 and e2), in the bouncer, to set-reset Flip-Flops. Gate level implementation of one stage of the tracker and the bouncer is shown in Figure 14. Note, only one out of four *set functions* of the latch is shown here. This set function will have four, three-input AND gates, whose outputs are Ored and then inputted to the NOR gate in the latch.

Simulations results for this checker are similar to those shown in Figure 11, in which signals e1 and e2 generate code words during the fault-free behaviour and non-code words for faulty operation.

6 Comparison of checker designs

In this section, the checker designs obtained by Petrify and Direct mapping are compared. Checkers are compared in terms of area, latency and testability. The results of the comparison are shown in Table 1.

Table 1: Comparison of Checker implementation

Checker	Petrify	Direct mapping
Area (μm^2)	853	2516
Latency (ns)	1.13	0.84
Testability (%)	53	78

Circuits obtained by Direct mapping are larger in size compared to those obtained by Petrify. However,

these circuits are better in terms of latency and testability. These circuits are 100% (off-line) testable as shown in [20]. But the on-line testability of these circuits is low, because of the gates that implement the refusal sets. These gates are used in the set function of the bouncer latches and are activated only when the protocol is violated. Under normal operation (no protocol violation), these gates do not affect the outputs (e1 and e2). Hence, a fault in these gates is not detected. Also these gates contribute to the larger size of the checker.

7 Conclusion

A method of concurrent error detection at asynchronous interfaces has been explored. Checker designs for handshake protocols are implemented in AMS-0.35μ process using Cadence tools and simulated. The checkers detect out-of-order errors in the protocol under test. They are based on busy signalling approach, in which the output is constantly switching under normal operation.

The enumeration of the checker faults has shown the coverage of 53% and so a new checker based on direct mapping approach is proposed. The two checker designs are compared in terms of area, latency and testability.

In future we would like to improve the self-testability of checkers. The practicality of the proposed method still needs to be assessed. Asynchronous checkers would require a significant amount of area overhead when used for multiple handshakes. The designers should apply this technique selectively at appropriate level of the system design, e.g. this could be cost effective when used in GALS systems where large portions of system are tested by traditional means.

References

[1] http://public.itrs.net/homestart.htm

[2] E. Dupont, M Nikolaidis, P. Rohr, Embedded robustness IPs for transient-error-free ICs, ICCC Design and Test of Computers, Vol 19, No 3, 2002, pp 56-70

[3] R. Dobkin, R. Ginosar and C. P. Sotiriou, Data Synchronous Issues in GALS SoCs, Proc. Of International Symposium on Advanced Research in Asynchronous circuits and Systems, pp 170-179, IEEE Computer Society Press, April 2004

[4] Y. S. Dhillion, A. U. Diril, A. Chatterjee and A. D. Singh, Sizing CMOS circuits for increased transient fault tolerance, Proc. of 10th IEEE International On-line Testing Symposium, pp 11-16, Portugal, July 2004

[5] M. Nicolaidis, Y. Zorian and D. K. Pradhan, On-line testing for VLSI, Kluwer Academic Publishers, 1998

[6] Parag Lala, Self-Checking and Fault-Tolerant digital Design, Academic Press, ISBN 0-12-434370-8, 2001

[7] P.D. Hyde and G. Russell, A Comparative Study of the Design of Synchronous and Asynchronous Self-Checking RISC Processor, In the Proc. of 10th IOLTS'04, pp 89-95, Funchal, Madeira Island, Portugal, July 12-14, 2004

[8] Victor I Varshavsky, M. Kishinevsky, V. Marakhovsky, et. al., Self-Timed Control of Concurrent Processes, Kluwer Academic Publishers, The Netherlands, 1990

[9] I. David, R. Ginosar and M. Yoeli, Self-Timed is Self-Checking, Journal of Electronic Testing: Theory and Applications, 6, April 1995, pp 219-228

[10] Alain J. Martin, and P. Hazewindus, Testing Delay-Insensitive Circuits, MIT, 1991, pp 118-132

[11] J. Sparso, S. Furber, Principles of Asynchronous Circuit Design: A System Perspective, Kluwer, 2001

[12] A. Bystrov, A. Yakovlev, Asynchronous Circuit Synthesis by Direct Mapping: Interfacing to Environment, Proc. of Async, Manchester, April 2002, Pages 127-136

[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, {Petrify}: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers, XI Conf. on Design of Integrated Circuits and Systems, Barcelona, Nov. 1996

[14] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, A region-based theory for state assignment in speed-independent circuits. IEEE Transactions on Computer-Aided Design, 16(8): 793-812, August 1997

[15] U. Goltz, W. Reisig, P. S. Thiagarajan, Two alternative definitions of Synchronic Distance. Informatik-Fachberichte 52: Application and Theory of Petri nets: Selected papers from the First and Second European Workshop on Application and Theory of Petri nets, Strasbourg, Sep 23-26, 1980, Bad Honnef, Sep 28-30, 1981, editors Girault, C. and Reisig, W. Springer-Verlag, 1982.

[16] F. Commoner, A. W. Holt, S. Even, A. Pneuli: Marked Directed Graphs. Journal of Computer and System Sciences vol. 5, Pages 511-523, 1971

[17] L. Y. Rosenblum and A. V. Yakovlev, Signal graphs: from self-timed to timed ones, Proceedings of International Workshop on Timed Petri Nets, IEEE Computer Society Press, Torino, Italy, pages 199-207, July,1985

[18] I. E. Sutherland, and J. Ebergen, Computer with Clocks, Scientific American, July 2002

[19] M. Kishinevsky, A. Kondratyev,A. Taubin and Victor Varshavsky, Concurrent Hardware: The Theory and Practice of Self-Timed Design, Wiley, 1994

[20] D. Koppad and A. Bystrov A. Yakovlev, Off-line testing of Asynchronous circuits, 18th Intl. Conf. on VLSI Design, IEEE CS Press, Kolkata, Jan 3-7, 2005

[21] Gordon Russell and Ian Sayers, Advanced Simulation and test methodologies for VLSI design, London: van Nostrand Reinhold (international), 1989

[22] A. V. Yakovlev, A. M. Koelmans, A. Semenov and D. J. Kinniment, Modelling, analysis and synthesis of asynchronous control circuits using Petri nets, Integration, the VLSI journal, vol 21, Dec 1996

[23] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985

[24] A. Bystrov, A.Yakovlev, Synthesis of Asynchronous circuits with predictable latency, 11th International Workshop on Logic & Synthesis, Louisiana, USA. IEEE Computer Society, June 4-7, 2002

[25] Hollar, Lee A, Direct Implementation of Asynchronous Control Units, IEEE Trans on Computers, vol. C-31 Dec 1982

[26] R. David, Modular design of asynchronous circuits defined by graphs, IEEE Trans on Computers, Aug 1977

[27] Al Davis and S. M. Nowick, An Introduction to Asynchronous Circuit Design, Dept. of Computer Science, University of Utah UUCS-97-013, Sept 1997

[28] A. Bystrov, D. Sokolov, A. Yakovlev, Low Latency Control Structures with Slack, The Ninth IEEE International Symposium on Asynchronous Circuits and Systems, Vancouver, 12-16 May 2003.