2009 15th IEEE Symposium on Asynchronous Circuits and Systems

Modular approach to multi-resource arbiter design

Stanislavs Golubcovs, Delong Shang, Fei Xia, Andrey Mokhov, Alex Yakovlev
{Stanislavs.Golubcovs, Delong.Shang, Fei.Xia, Andrey.Mokhov, Alex.Yakovlev}@ncl.ac.uk

***Abstract*—When circuits need to be constructed out of several self timed parts that access shared resources, asynchronous arbitration is often required. We consider the creation of the general purpose arbiter delegating M resources to N clients for the cases when the resources can be either active or passive participants of the arbitration. Firstly, the problem is solved for the case of two active resources being offered to two clients. Then a general problem solution is provided. Finally, on the basis of the 2×2 arbiter designed, it is shown how it can be used to create a scalable multi-resource arbiter for passive resources.**

I. Introduction

There are a number of arbitration examples (many-to-one and many-to-many) that can be found at all levels of computing systems. For instance, in software, the processes in concurrent environments are synchronised using the semaphores. The code accessing some common resource is marked as critical section. Before a process may execute the code, it checks the semaphore value to find out, whether the code is executed by any other process and locks it after entering the critical section.

Another mathematical problem is the famous problem of the "Dining Philosophers". Each philosopher sitting around a round table either eats or thinks. To eat, a philosopher needs to take forks in his left and right hands, each of which is shared with his left and right neighbours. When all philosophers start by taking the forks in their right hands first, no one will manage to take the second fork and the system will be stalled in a deadlock state. Obviously, this is a system of two-to-one arbiters, where clients are the philosophers and forks are the shared resources.

One life example could be the elevator lifting not more than a certain amount of people. This type of resource is able to serve several clients simultaneously; however, it still has some limiting factor, preventing all clients from being served at once. In theory, it would relate to the so called "multi-token" arbiters, where each client receiving a grant is taking away one or maybe several tokens, and bringing them back, when the resource is not used. As long as there are tokens left, new clients may access the resource. Practically, such an arbiter would create an artificial bottleneck for the device throughput, and help to reduce the impact of the bursty signal environment.

A. Arbitration in circuits

In many cases, and particularly in the area of systems-on-chip, one could think of complex resource allocation implemented in hardware. When there are several resources capable of providing the same kind of service for any of the clients, the arbiter's task would be to choose one of the resources available and offer it to a client making a request. Resource examples could be processors executing client instructions, or data transmission channels or any other example, where the arbiter would distribute a set of interchangeable resources over a number of client requests.

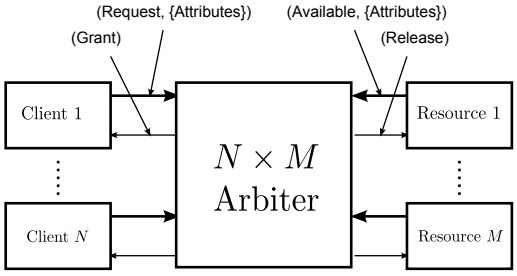


Figure 1. Synopsis of an arbiter (picture reproduced from [1])

From a general point of view, an arbiter manages clients accessing one or several resources (Figure 1). The communication with the arbiter is done via two-way communication channels. Channels are formed by the request/acknowledgement pairs of signals. Regardless of whether two- or four-phase protocol is used [2], every client needs a way of telling when a resource is needed, and every resource also needs some way of telling that it is available for clients. Sometimes the request part of both resource and client channels can provide additional information, which can be utilised by the arbiter and can influence its behaviour. For example, these requests can hold information about the priority or the "quality" of the resource, which may affect arbiter decision among several available grant scenarios. (In the case of two-phase protocol, the communication between a client and the arbiter is often based on three signals (request, grant, done)).

In such a general kind of arbitration task we assume resources being the *active* members, enabling the arbitration. The arbiter does not necessarily need to know whether some of the clients are actually using the resource because the knowledge of the resource availability is specified explicitly. A resource may not become available immediately after a client finishes using it and may require a non-specified time period to become ready again.

This type of arbiter can also be addressed in the context of the *Committee Problem*, where in general case various combinations of professor requests can start committees. There are several solutions proposed in [3]. The solutions are based on high-level 2-phase models. The same approach could be adapted for our case, if we assumed the initial requests to be the professors and a connection established between an outgoing grant pair (which we call *handshake*) to be the starting

1522-8681/09 $25.00 © 2009 IEEE
DOI 10.1109/ASYNC.2009.16

event of a committee. However, the solutions given in [3] use either explicit or implicit polling mechanisms and/or multi-way arbiters which, together with the 2-phase assumption, do not lead to efficient hardware implementations. In addition, the solutions are not supported by a formal verification process which may be a problem especially for the more concurrent ones.

The majority of existing arbiters assume that every resource is always available, when not used by any of the clients. When one of the clients requests the arbiter needs to guarantee that the resource being offered is not used by any other client. In this case, the resource is *passive* because it does not participate in the arbitration. There is also a lack of formal analysis of these solutions.

In this paper we consider the creation of multi-resource arbiters working with both active and passive resource types. The main contributions of the paper are as follows:

- We provide speed-independent implementations based on fast four-phase protocols;
- We try to minimise critical section time and the overall response latency of the circuits;
- We formally describe the problem by using Petri nets and State Graphs which allows us to use synthesis tools and formally analyse the results;
- We show how a complex conflict can be subdivided into two simple conflicts which can be resolved by using basic 2-input MUTEX elements.

The structure of the paper is as follows:

- Section 2 describes the functionality of the active resource based arbiter. It first shows how the conflict is resolved for the small 2×2 arbiter and then demonstrates ways how such design could be scaled to support a greater number of clients and resources.
- Section 3 considers the simpler task of distributing M passive resources (not making requests) among N clients making requests. A scalable solution is obtained which consists of separate autonomous cells where each cell is based on the structure of the 2×2 arbiter described in Section 2.

II. Arbiter design

Since the arbiter granting M resources to N clients is a complicated design task, we first define the problem and try to solve it for a simple 2×2 case.

A. Functionality

Structurally, the active resource arbiter is symmetric from both sides, it has the same communication interface (we don't technically need to specify which side represents the client and which the resource). Its task can be rephrased as the activation of available handshakes between left and right neighbouring circuits effectively pairing outstanding requests from different sides (Figure 2). Both clients and resources actively participate by producing requests (clients request when they need a resource and resources make requests to inform the arbiter about their availability).

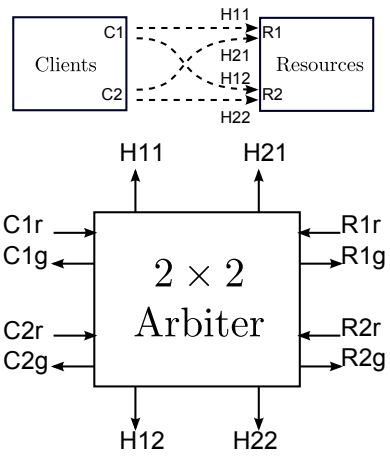


Figure 2. 2×2 arbiter interface

When at least one resource is available and one client is requesting it, a handshake is possible. It results in an arbiter activating the corresponding channel and informing both the client and the resource by using outgoing grant signals. The communication direction can proceed from client $C1$ or $C2$ to resource $R1$ or $R2$. The identification of the connection is provided by outgoing handshake grants $H11$, $H12$, $H21$, $H22$ (also called channels). Naturally, the grant on channel $H11$ will consequently produce grants $C1g$ and $R1g$ for client $C1$ and resource $R1$. As soon as the client (or the resource) has received the grant signal, it can be sure that the other participant is also ready and waiting. After this point, the client (or the resource) can signal the arbiter that the job is done by removing its request (signals $R1r-$ and $C1r-$ are concurrent and may have different timing for processing the grants). The removal of the request will eventually result in the arbiter removing the grant as well. The grant signal issued by the arbiter is persistent. The arbiter waits until both sides remove their requests, and only then simultaneously removes their grants. For the correct functioning of the circuit, both sides have to wait until the grant is released before the preparations for the next request.

We model the behaviour of the arbiter by means of Signal Transition Graphs (STGs) which are a special type of Petri Nets, whose transitions are interpreted in terms of signal transitions (i.e. rising and falling edges, denoted by $X+$ and $X-$ respectively) [4].

As it can be seen from the STG diagram (Figure 3), the channel activation has a behaviour with mutual exclusion. When, say, client $C1$ is busy communicating with $R1$, it is not available for other channels using either the first resource or the first client. Hence, the activation of $H11$ will prevent activation of $H12$ and $H21$. Similarly, $H21$ would disable $H11$ and $H22$, $H12$ would disable $H11$ and $H22$, and $H22$ would disable $H12$ and $H21$.

It is important that the circuit does not prevent concurrently enabled handshakes. When all four requests come at the same

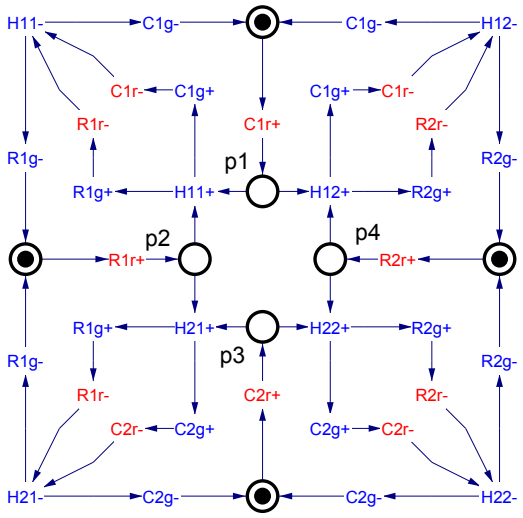


Figure 3. 2×2 arbiter STG

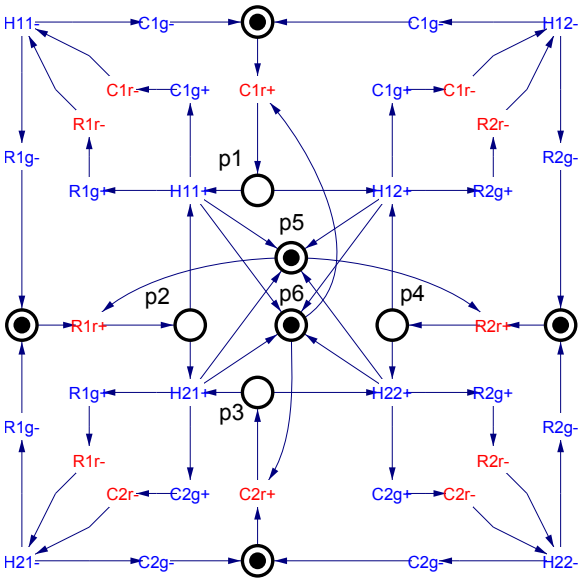


Figure 4. Additional exclusion places added

time, the arbiter makes a decision and connects requests by activating two non-conflicting handshakes (which are $H11$ and $H22$ forming parallel connections or $H12$ and $H21$ forming the over-crossing connections).

B. Resolving the conflict

The STG presented in Figure 3 contains 8 conflicts (4 conflict pairs): $H11 \longleftrightarrow H12$, $H11 \longleftrightarrow H21$, $H22 \longleftrightarrow H12$, and $H22 \longleftrightarrow H21$. It is not clear how such a complex conflict could be resolved on a level of transistors. It is known from the literature [1], [5] that building a complex arbitration structure (more complex than a 2-way MUTEX) at a transistor level is not advisable because such structures may be prone to oscillatory behaviour. Hence, we pose here a problem of creating an architecture resolving the conflict in which only 2-way mutual exclusion elements are used as basic arbitration components.

Let's assume that client requests ($C1r$ with $C2r$) and resource requests ($R1r$ with $R2r$) are mutually exclusive. Hence, only one handshake signal can fire at a time. Such change moves the conflict from places $p1$, $p2$, $p3$, $p4$ to places $p5$ and $p6$ (Figure 4). The state graph produced from the STG diagram shows that the concurrent transmissions are still possible although the activation of the handshakes is sequential (Figure 5).

Since signals $C1r$, $C2r$, $R1r$, $R2r$ are input transitions (Figure 4), we cannot change conditions activating them because it would mean a different communication protocol with the environment and would not be consistent with the initial specification (Figure 3). It is still possible, however, to change the initial STG by inserting additional internal signals ($rc1$, $rc2$, $rr1$, $rr2$, $gc1$, $gc2$, $gr1$, $gr2$) that would control the handshakes and ensure the desired behaviour of their mutual exclusiveness.

Such a refinement is depicted in Figure 6. All of the handshakes are controlled by the incorporated MUTEX element

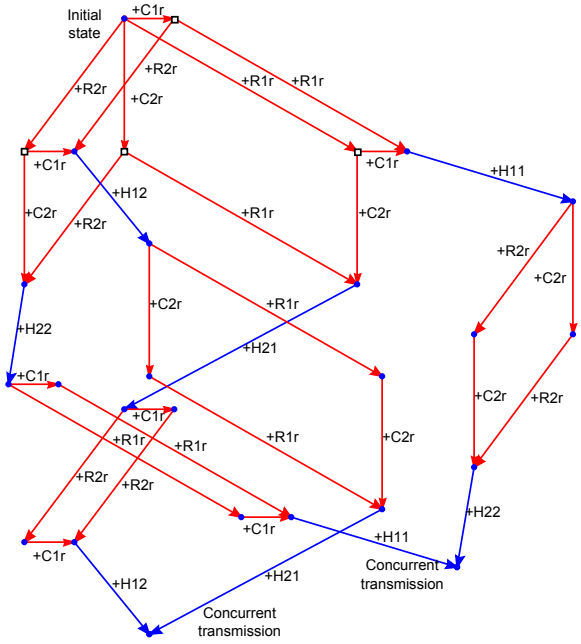


Figure 5. State graph of the SET phase of the modified STG

STGs with their input request signals $rc1$, $rc2$, $rr1$, $rr2$ and output grant signals $gc1$, $gc2$, $gr1$, $gr2$. We treat these MUTEX elements as the environment, therefore, the grant signals are inputs in our model (Figure 6).

This diagram can still be shaped to find a compromise between the speed and the complexity of the circuit and also make it possible to create the circuit by using methods described in [4] and use an automated tool such as *Petrify* [6]. For instance, we may add additional connections from $gc1- \rightarrow H11-$ (and others as shown with dashed arrows

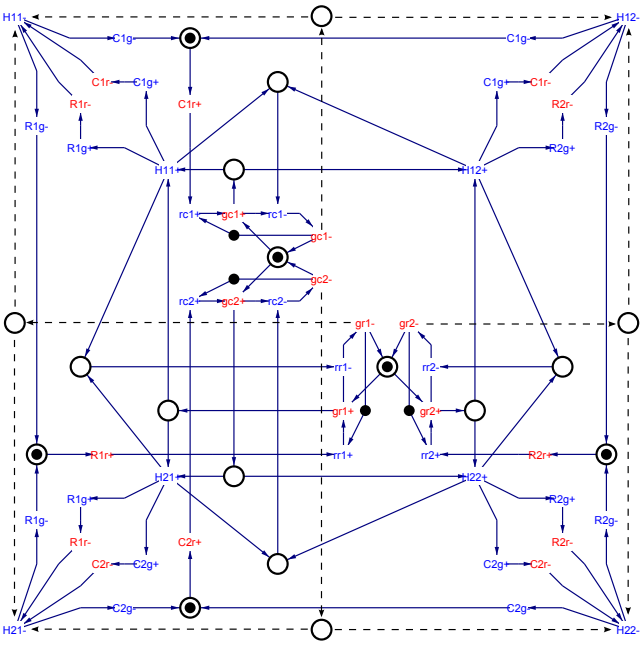


Figure 6. STG with MUTEX elements

in Figure 6). This modification would eliminate all Complete State Coding conflicts [4]. We would assume that such a concurrency reduction would not really make the circuit slower because the path $C1g+ \rightarrow C1r-$ contains the state of the client actually accessing the resource which is expected to be slow, and by that time $gc1-$ would definitely fire and give its token to $H11-$.

If the model is given to *Petrify*, the following output is obtained:

$$
\begin{aligned}
C1g &= H11 + H12 \\
C2g &= H21 + H22 \\
R1g &= H11 + H21 \\
R2g &= H12 + H22 \\
H11 &= gc1 \cdot gr1 \cdot \overline{H12} \cdot \overline{H21} \\
&\quad + H11 \cdot (gc1 + gr1 + C1r + R1r) \\
H12 &= gc1 \cdot gr2 \cdot \overline{H11} \cdot \overline{H22} \\
&\quad + H12 \cdot (gc1 + gr2 + C1r + R2r) \\
H21 &= gc2 \cdot gr1 \cdot \overline{H11} \cdot \overline{H22} \\
&\quad + H21 \cdot (gc2 + gr1 + C2r + R1r) \\
H22 &= gc2 \cdot gr2 \cdot \overline{H12} \cdot \overline{H21} \\
&\quad + H22 \cdot (gc2 + gr2 + C2r + R2r) \\
rc1 &= \overline{H12} \cdot \overline{H11} \cdot C1r \\
rc2 &= \overline{H22} \cdot \overline{H21} \cdot C2r \\
rr1 &= \overline{H21} \cdot \overline{H11} \cdot R1r \\
rr2 &= \overline{H22} \cdot \overline{H12} \cdot R2r
\end{aligned}
$$

C. Implementation

The arbiter can be constructed out of several asynchronous latch types such as C-elements and MUTEX-es.

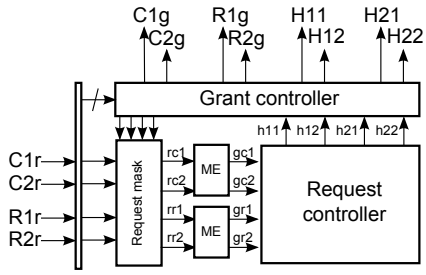


Figure 7. Arbiter structure

C-element: The basic, two input C-element introduced by Muller [7] has a behaviour described by the formula:

$$Q' = A \cdot B + (A + B) \cdot Q.$$

It means that the output of the gate is set when both inputs are active, it is reset when both inputs are inactive, and it stays unchanged when inputs are different.

The design described later in this paper uses three input C-elements:

$$Q' = A \cdot B \cdot C + (A + B + C) \cdot Q$$

with three signals activating and deactivating the output; and the asymmetric four-input C-elements:

$$Q' = A \cdot B \cdot C \cdot D + (A + B) \cdot Q,$$

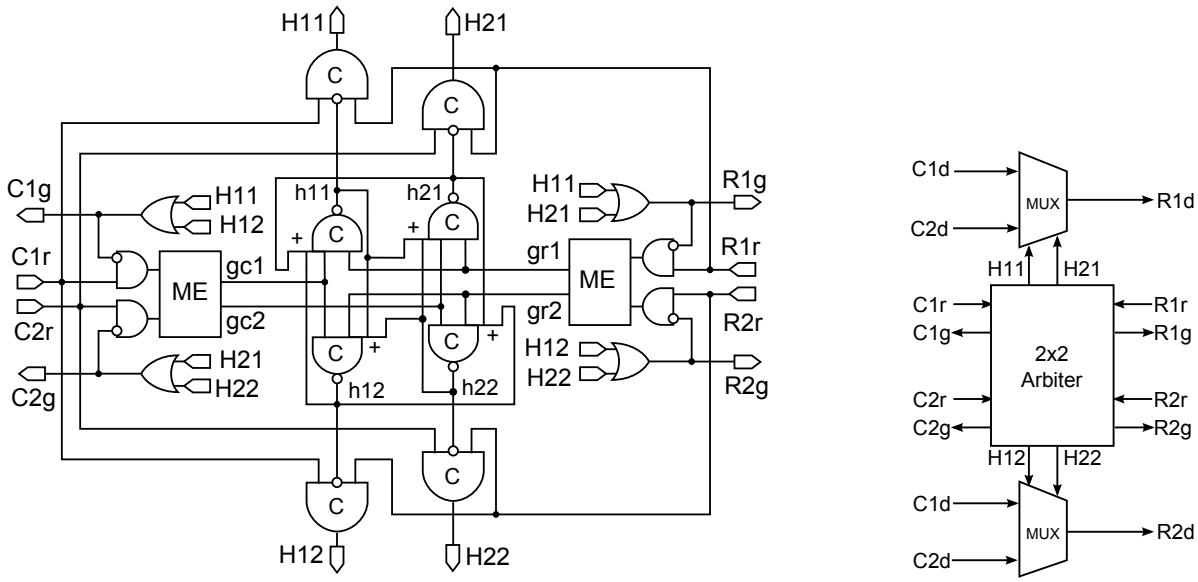
where all four signals are used to activate the output and only two signals (A and B) used to deactivate it.

MUTEX element: MUTEX is the basic two input arbiter [8], [9], often used as a construction block for arbiters of a more complicated structure. It is implementable as a couple of NAND gates with a metastability resolver. The arbiter task is to ensure the correct dual-rail output (when not more than one output signal can be active). When both inputs rise, it hides the associated metastability and waits until one of the signals eventually resolves it.

Arbiter implementation: The design implementation is subdivided into separate functional parts. By introducing new internal signals $h11$, $h12$, $h21$, $h22$ we subdivide the implementation of signals $H11$, $H12$, $H21$, $H22$ into the Grant controller and the Request controller which allows us to use simpler gates. As a result, we present one of possible decompositions for the equations provided by *Petrify*.

Initially, all requests from both sides can arrive at any moment. Suppose that $C1$ has issued a request. The signal first propagates through the request mask and then it is being arbitrated with neighbouring request channel $C2$ using ME element (Figure 8a). Both ME elements will ensure that there are at most one client and one resource entering the request controller part. This eliminates the choice of the handshake that is always possible when three or four requests arrive at the same time.

Suppose at some point signals $C1r$ and $R1r$ win the arbitration. As soon as two arbitrated requests $gc1$ and $gr1$



(a) Implementation schematics

(b) Usage example

Figure 8. 2×2 arbiter

following MUTEX-es arrive, they are transformed into a request for the channel $H11$. When it happens, the grant controller activates the channel and provides grant signals to its associated client and server. From this moment, the central part of the circuit has fulfilled its function and can be reused to work with other requests, potentially forming the second, concurrent handshake. In order to allow other requests to enter the request controller, both MUTEX elements should be released. This is done by connecting corresponding grants with the request mask. The mask consisting of the AND gates "hides" the initial requests, leaving MUTEX-es free, and allowing signals $gc2$ and $gr2$ to enter. When the request controller is unblocked, it will be able to activate other channels that do not conflict with $H11$ (in our 2×2 case it can only be the channel $H22$).

Request controller: The speed-independent implementation of the request controller consists of four asymmetric C-elements producing the inverted channel request signals. It establishes the channel request (i.e., changes it from 1 to 0) when a pair of arbitrated requests arrives and removes it when both requests are removed. Without explicit delay assumptions, request masks activated by the same channel grant can react differently and free the left and right MUTEX-es at distinct moments of time. As soon as either of the ME elements is freed, it lets the new request come in. If only one MUTEX, say, the one arbitrating $C1r$ and $C2r$, has switched grants from $c1$ to $c2$, while the second MUTEX didn't manage to remove the grant $r1$, for a short period of time the request controller has $c2$ and $r1$ as the candidates for the next channel activation. This is a hazard, because channel $H11$ has been activated and $r1$ cannot be used for other handshakes. This hazard is prevented using additional inputs on the set phase of C-elements.

Grant controller: The grant controller consists of four three-input C-gates and four two-input OR gates. Each channel is activated by synchronising the requests from the associated participants and the selection of the request controller. The activated channel then propagates through the OR gates to deliver the grant signal to the initial requestors. The channel signals can be used to select the right data propagation path as shown in Figure 8b, which is then followed by the outgoing grant signal activating the access.

D. Verification of the circuit

The circuit can be formally verified using command line tools *Punf* [10], [11], *MPSat* [12] and *Workcraft* integrated environment as described in [13]. First, the gate-level model of the circuit is created in *Workcraft*. Then it is converted into its Petri net equivalent (the circuit Petri net). The model produced defines the causality of the output transitions. However, it does not have information about the input transitions, which is being defined in a separate interface Petri net diagram (also called the environment model of the circuit). The environment information is present in the initial STG model (Figure 3), showing that, for example, the input transition $C1r-$ occurs after the output transition $C1g+$. The circuit Petri net combined with the environment is saved as a `*.g` file. Then the unfolding tool *Punf* is used to unfold it instead of analysing the state graph. The tool produces a `*.mci` file which is smaller and easier to analyse compared to the full reachability state graph. Finally, the *MPSat* tool is used to verify whether certain conflict states of the Petri net are possible. Such conflict states involve a choice among two or more transitions, when one transition fires, the second is disabled. If there are no output transitions being disabled by any reachable state, then the circuit produced

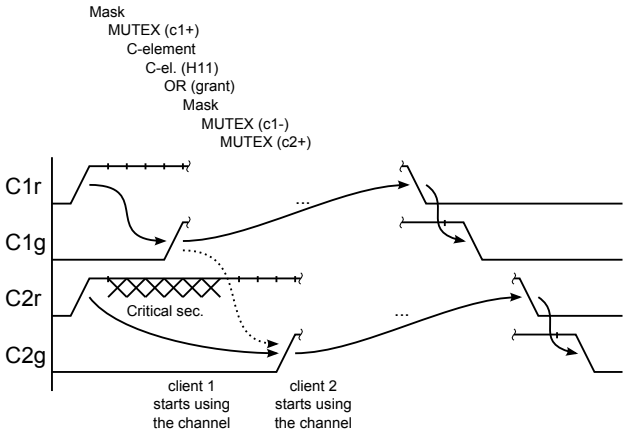


Figure 9. Timing diagram

by such a Petri net is *speed-independent* and does not contain *hazards* [13].

The implementation of the circuit was verified using the method described above. The output of *MPSat* confirmed that there are no reachable states (which are present in the initial STG (Figure 3)) where one of the handshakes disables the activation of the others. It was also checked that the states activating concurrent non-conflicting handshakes ($H11$ and $H22$ or $H12$ and $H21$) are reachable and there are no reachable states activating the conflicting handshakes.

E. Latency estimation

Because the channel controller is only capable of processing one handshake at a time, it has a critical section delaying the grant response time to the client being temporarily blocked by MUTEX. We can estimate the worst and the best case scenarios for the arbiter response time, when both client requests arrive at the same time and the resources are always available and do not introduce additional delays (Figure 9). Of course, the response time is dependant on both client and resource sides; however, they are processed in parallel and the arbiter delay can only be estimated from the moment the combination of requests that actually allows the handshake.

For simplicity and comparability assume that all gate delays are the same and there is no additional delay due to metastability in the MUTEX-es. In the beginning, both requests propagate through the request mask simultaneously and arrive into the MUTEX. The MUTEX reacts by allowing only one client in and creates the critical section part delaying the second client. In our example $C1r$ wins the MUTEX arbitration and will receive the handshake in the first place and give it a better response time.

The second client waits until MUTEX is freed by the request mask. Since $C2r$ has arrived together with $C1r$, it needs to wait the full amount of the critical section delay. Critical section introduces six gate delays making the worst case response time to be eleven gate delays. As it can be seen from the picture, the true benefit from using parallel handshake architecture can be achieved only when the channel needs to be occupied considerably longer than the delay of the critical section. It can also still be a safeguard if the client reaction time for $C1g+ \rightarrow C1r-$ is not known in advance and may take various times.

F. Simulation in Spectre

The circuit was modelled in the Cadence environment, using UMC 90 nm technology libraries and then simulated using the Spectre analogue analysis tool. Table I lists the results.

Note: The rising and falling edge of any of the inputs is set to 50 picoseconds and the supply voltage is 1 volt. All of the measurements are taken when the edge hits 0.5V threshold. Hence, when the delay of the initial request is 0, its rising edge reaches the threshold after 25 picoseconds. The response latency for the first example in Table I can be estimated as $449 - 25 = 424$ picoseconds.

The first columns (from $C1r$ to $H22$) show the absolute timing of arriving and outgoing signals. The following two columns show the time that has passed since the moment the request was initiated (that is the moment when both client and resource arrive and the circuit actually can respond with at least one handshake activation). In the second and the third examples the response latency for the first ($H11$) and the second ($H22$) channels is greater because of the metastability created by the conflicting requests arriving closely. In the fourth example, the request for the second channel ($H11$) arrives 150 picoseconds after the circuit has already started activating the first channel ($H22$). As we can see, the arbiter is still busy with the first pair of requests; however, the second pair needs to wait 150 picoseconds less till the moment the channel is open.

Finally, in the last example (row 5) requests arrive one-by-one, without creating conflicts. Second pair does not overlap with the first pair and it shows the best response time.

G. Extending up to $N \times M$ arbiter

Theoretically, it is possible to extend the existing design up to N clients and M resources. In other words, we would need to activate $N \times M$ different handshakes. An example of 4×3 arbiter is given in Figure 10, where the internal requests $r1g$, $r2g$, $r3g$ control rows and $c1g$, $c2g$, $c3g$, $c4g$ control columns. The 3- and 4-input arbiters delivering requests into the request controller. Similarly to the 2×2 arbiter all conflicting neighbours need to be disabled before the initial requests are masked to let new requests propagate into it. In particular, the handshake row needs to be disabled before the new client request and the handshake column before the new resource request.

One possible decomposition of the cell is shown on the left side of the picture (Figure 11a). As soon as the resource and the client grant come in, they lock the other five conflicting neighbours and outputs handshake signal H_{ij}. Since only one handshake can happen at a time, the outgoing grant signals C_ig can be activated by OR-ing the i-th column handshakes

Table I
RISING EDGE TIME ESTIMATION IN SPECTRE

#	$C1r$(ps)	$C2r$(ps)	$R1r$(ps)	$R2r$(ps)	$H11$(ps)	$H12$(ps)	$H21$(ps)	$H22$(ps)	1st lat.(ps)	2nd lat.(ps)	2nd−1st(ps)
1	25	—	25	—	449	—	—	—	424	—	—
2	25	30	25	30	505	—	—	962	480	932	452
3	25	26	25	26	535	—	—	992	510	966	456
4	25	275	275	125	—	538	994	—	413	719	306
5	625	25	675	75	1088	—	—	488	413	413	—

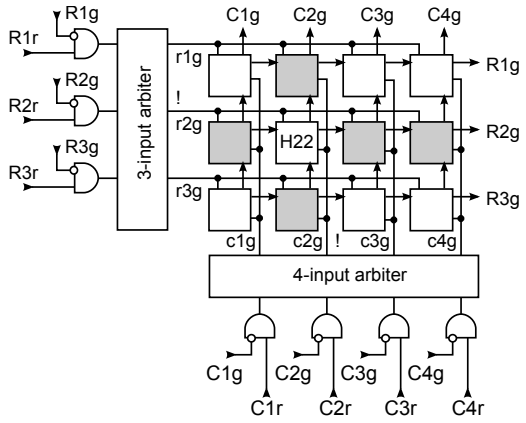


Figure 10. Schematics for the implementation of 4×3 arbiter.

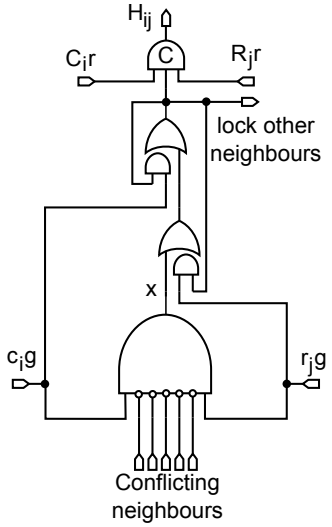
and similarly the resource grants R_jg are calculated by OR-ing the j-th row handshakes.

The NOR gate in the cell implementation has to have $N + M - 2$ inputs, which is not practical for $M + N > 4$. To solve this problem, we either need to decompose it into smaller gates and introduce timing assumption that there are no glitches on the x signal or alternatively develop the block mechanism that would explicitly disable all the conflicting neighbours before the outgoing handshake signal occurs.

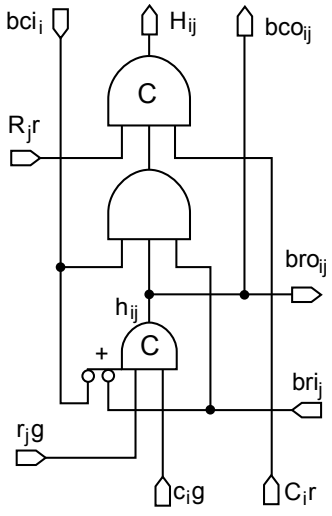
As we can see from the schematics in Figure 10, handshakes of the same column or row are always in conflict, which leads to the idea of creating signals blocking each row and column separately. An example of such a cell is shown in Figure 11b. The internal grants r_jg and c_ig can only activate the internal handshake h_{ij} when if it not disabled by the column and the row block signals bci_i and bri_j. Once activated, h_{ij} issues outgoing block requests bco_{ij} and bro_{ij}. The signals bci_i and bri_j are obtained by OR-ing all column bco and row bro. It is known that only one internal handshake can be active at a time. It means that these OR elements can be safely decomposed into a tree of 2-input OR gates. By the time bci_i and bri_j arrive, the internal handshake h_ij can be propagated further and enable the external handshake H_{ij}. A broader review of different cell implementations is provided in [14].

H. Latency estimation

A disadvantage of such a solution is the linear growth of the worst case responce latency. The worst delay can be estimated as: $L(N, M) = (\min(N, M) - 1) \cdot \delta$, where δ is the critical section delay consisting of the time for the signal to pass the



(a) Extending 2×2 arbiter cell for 5 neighbours



(b) Separate column and row block signals

Figure 11. $N \times M$ arbiter cell implementation

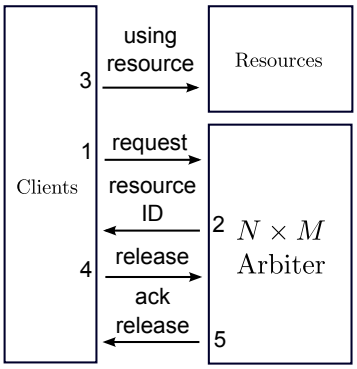


Figure 12. Asymmetric multi-resource arbiter structure

N- or M-input arbiter, through the handshake grid and return the initial request back to mask.

An alternative approach would be to use a *lock* signal taking a snapshot of current input states in a similar way as it is done in priority arbiters [15]. The lock mechanism formed by a column of the MUTEX elements would allow the granting of all the handshakes simultaneously, in the optimistic case, reducing the circuit response latency to a constant. The comparison of these two approaches is the subject for future research.

I. Fairness of the arbiter

Along with the introduction of more clients and more resources there is a possibility of more than one request waiting its resolution which raises a question about the fairness of the arbiter among client or resource requests. It is, however, easy to see that the fairness of the arbiter is dependant on the fairness of the internal arbiters delivering the requests to the request controller. We also may use different priority schemes on both sides of the arbiter, for instance, priority arbiter for the clients and at the same time token ring or the ordered arbiter for the resources.

III. Multi-resource arbiter for passive resources

A. Task specification

In the remaining part of the paper we consider the creation of a multi-resource arbiter managing access to a number of passive resources. For a given client request, the arbiter would return the available resource and it would make sure that no more than one client is receiving the same resource (Figure 12).

After the resource is no longer required, the client needs to inform the arbiter that it is releasing the resource, so that the arbiter could offer the same resource to other clients. Depending on the protocol, we also could have an explicit signal to acknowledge client that the resource was released.

A similar kind of arbiters was presented earlier in [16], where several passive resources are being distributed among clients. The solution, however, does not scale well as the number of N-way and M-way arbiters is linearly increasing. Nevertheless, the work inspires to model both types of arbiters and compare their performance.

We propose the implementation based on the idea of the multi-token ring arbiters presented in [17]. The token ring would consist of a certain number of separate cells, each cell connecting to one client and two neighbouring cells (Figure 13). The tokens propagating inside the ring are considered as available resources and can be captured by clients when they need resources and later inserted back into the ring.

As we see, the model of such a cell is similar to the 2×2 arbiter described earlier in this paper. The former client side requests $C1$ and $C2$ would correspond to the propagation channels 'Token put' (releasing the token back into the loop) and 'Token in' (delivering token from the left neighbour). Alternatively, channels 'Token get' (capturing the token for the client) and 'Token out' (delivering the token to the right ring cell) would correspond to the resource side requests $R1$ and $R2$. Note, that now handshakes are formed between token propagation channels.

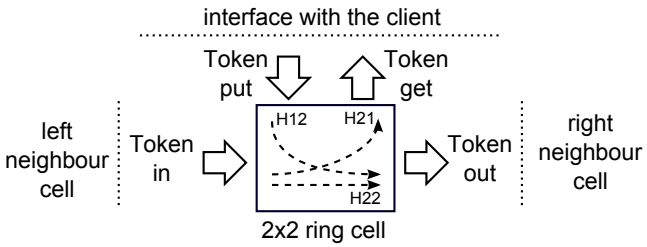


Figure 13. Busy token ring cell

The token events are asynchronous and can happen at any moment, except that, normally, a client would not try to release the token and try to get it at the same time. This leaves us with only three different token propagation scenarios. Firstly, the token can propagate through the cell from the left to the right neighbour (handshake $H22$ in the initial design). Secondly, it can be captured by the client, connecting channels 'Token in' and 'Token get', which would be the handshake $H21$. And finally, token can be released by connecting channels 'Token put' and 'Token out', the handshake $H12$. $H11$ is not needed anymore, which simplifies the circuit design. First of all, it means that there would be no simultaneous transfer of $H11$ and $H22$. Secondly, since the client either takes a token or puts it back into the system, there would be no simultaneous transfer of $H12$ and $H21$. This leads us to a simplified STG diagram (Figure 14).

The analysis of the diagram shows that we have two pairs of conflicts in this diagram: $H22 \longleftrightarrow H12$ and $H22 \longleftrightarrow H21$. The introduction of arcs connecting transitions $Tgr+$ and $Tpr+$ means that a client will consequently try to put and get tokens in order. That constraint, however, needs be omitted in case we would like to use clients requesting or releasing several resource tokens in one go (that special kind of client could be used to initially saturate the ring with tokens, or remove tokens from the ring). In our design, we consider the implementation not having this constraint.

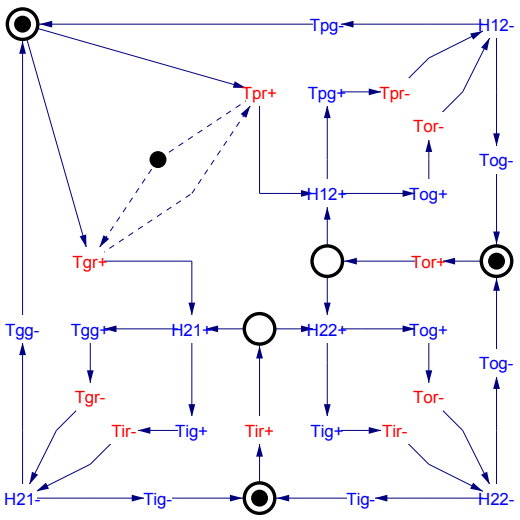


Figure 14. The STG of a ring cell

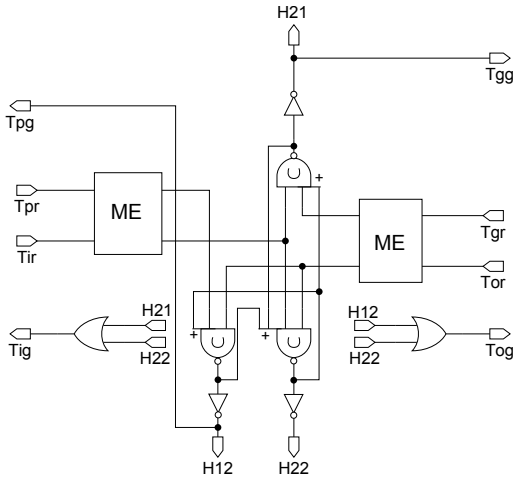


Figure 15. Implementation of the ring cell

B. Implementation of the ring cell

To implement the cell, we only need to take the existing 2×2 arbiter and simplify it. First of all, the absence of concurrent transfers removes the need for the gates masking initial requests. Additionally, because there are no concurrent transfers, we would not need C-elements forming the grant controller, as it can be done by using only the internal asynchronous C-elements forming the request controller in the initial design. We implement the request controller in the same way, except that the handshake $H11$ is not present. The handshakes still need to be gated in order for the circuit to work, because, according to the protocol, before the arbiter activates a new handshake, it needs to wait until both parties remove the request from the previous communication (Figure 15).

C. Implementation of the client controller and token controller

Apart from the cell providing the handshakes, two additional controllers are needed: the token controller acquiring the tokens from the left neighbour and delivering them to the right neighbour, as well as the client controller, providing the request/grant interface for a client (Figure 16). Each controller is associated with data register designed to store the identifier of the token.

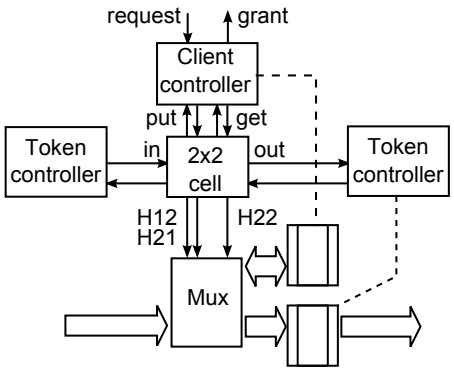


Figure 16. Ring cell structure

The behaviour of these controllers is shown in Figure 17. They can be implemented in a straightforward way, from the STG with *Petrify* [6], [4].

D. Latency estimation

Similarly to the usual token ring response latency, the response time of this arbiter would linearly increase as the number of clients N grows. Fortunately, this latency can be reduced if we let more resource tokens propagate through the ring. The more resource tokens are in the system, the less a client needs to wait till the moment it receives a token. We could estimate the latency as N/M, where M is the number of resources. For small M, response becomes linear, for large M, say, $M = N/2$, the average latency can be estimated as a constant value.

It should be noted that the final implementation might have a drawback for $M > N/2$. The large amount of resource tokens reduces time of a client getting a token, however, it would also reduce the number of free spaces which are needed to release unneeded tokens. This waiting time can cause the system to become slow again, as the release latency is estimated as $L(N, M) = \frac{N}{N - M}$. On the other hand, depending on system implementation, the token release wait may be put outside the critical path.

IV. Conclusions

This paper describes the asynchronous design of an arbiter managing handshakes between two clients and two resources. Each resource is actively reporting about its availability and can be connected to any of the clients. The initial STG of the circuit demonstrates the complexity of the conflict emerging among four incoming requests. One of the requirements ensured is the arbiter's ability to produce parallel, non-conflicting handshakes. To resolve the conflict, the original model is refined with a number of relatively fast internal transitions.

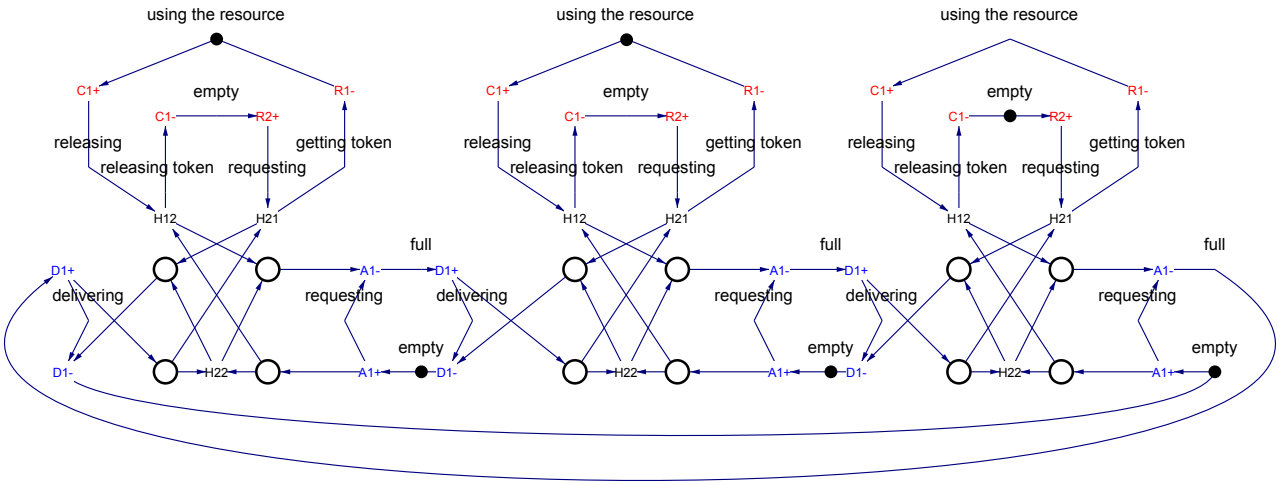


Figure 17. High level STG of a ring implementation with 3 clients and 2 resources

The activation of the handshake creating channels is therefore done in a sequential way but the user operation of the channels is concurrent.

A general solution for $N \times M$ arbiter is also described. It can be created for arbitrary number of inputs and is decomposable into simple gate elements.

Finally, the paper presents the way of creating a multi-resource arbiter with passive resources. For this easily scalable arbiter clients may receive grants in truly asynchronous manner.

During the initial design the *Petrify* tool was used to find implementations for separate circuit parts (such as the request controller and the grant controller), later it was also used to find different versions of the final design. Additionally, the overall circuit implementation was verified to be speed-independent using the *Punf* and the *MPSat* tools while all the modelling was done in the *Workcraft* environment.

Many opportunities are opened for future works. We have only covered $N \times M$ arbitration with four-phase (request-grant) protocol. It would be also interesting to find out what the two-phase protocol would look like as well as the arbiter 'Nacking' client requests when there is no available resource. As for the ring-based solution, it would be interesting to find and compare both busy and lazy ring implementations.

Acknowledgements: This work is supported by EPSRC GR/E044662/1. We would also like to thank A. Bystrov for helpful discussions about arbiters.

REFERENCES

[1] D. J. Kinniment, *Synchronization and Arbitration in Digital Systems*. John Wiley & Sons, Ltd, 2007.

[2] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design*. Boston/Dordrecht/London: Kluwer Academic Publishers, ISBN: 978-0-7923-7613-2, 2002.

[3] I. Benko and J. Ebergen, "Delay-insensitive solutions to the committee problem," *Advanced Research in Asynchronous Circuits and Systems, 1994., Proceedings of the International Symposium on*, pp. 228–237, November 1994.

[4] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, *Logic synthesis of asynchronous controllers and interfaces*. Springer-Verlag, ISBN: 3-540-43152-7, 2002.

[5] C. Van Berkel and C. Molnar, "Beware the three-way arbiter," *Solid-State Circuits, IEEE Journal of*, vol. 34, pp. 840–848, June 1999.

[6] Petrify: http://www.lsi.upc.es/~jordic/petrify/petrify.html.

[7] D. Muller and W. Bartky, "A theory of asynchronous circuits," in *Proc. Int'l Symp. Theory of Switching, Part 1, Harvard Univ. Press*, pp. 204–243, 1959.

[8] R. C. Pearce, J. A. Field, and W. D. Little, "Asynchronous arbiter module," *IEEE Trans. Comput.*, vol. 24, no. 9, pp. 931–932, 1975.

[9] C. L. Seitz, "Ideas about arbiters," *Lambda*, vol. 1, pp. 10–14, 1980.

[10] Punf: http://homepages.cs.ncl.ac.uk/victor.khomenko/tools/punf/.

[11] V. Khomenko, *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, University of Newcastle upon Tyne, February 2003.

[12] V. Khomenko, M. Koutny, and A. Yakovlev, "Detecting state coding conflicts in stg unfoldings using sat," *Application of Concurrency to System Design, 2003. Proceedings. Third International Conference on*, pp. 51–60, June 2003.

[13] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev, "Automated verification of asynchronous circuits using circuit petri nets," *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, pp. 161–170, April 2008.

[14] F. X. A. M. A. Y. S. Golubcovs, D. Shang, "Multi-resource arbiter decomposition," tech. rep., Newcastle University, 2009.

[15] A. Bystrov, D. J. Kinniment, and A. Yakovlev, "Priority arbiters," in *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, (Washington, DC, USA), pp. 128–137, IEEE Computer Society, 2000.

[16] S. S. Patil, "Forward acting n x m arbiter," tech. rep., Computation Structures Group Memo 67, M.I.T., 1972.

[17] A. Yakovlev, "Designing arbiters using petri nets," *VLSI Systems Research Center, Israel Institute of Technology, Haifa, Israel*, pp. 179–201, 1995.