Multi-resource arbiter design

Stanislavs Golubcovs, Andrey Mokhov, Alex Yakovlev
{Stanislavs.Golubcovs, Andrey.Mokhov, Alex.Yakovlev}@ncl.ac.uk

***Abstract*—When circuits need to be constructed out of several self-timed parts, the arbitration is often required for the asynchronous design. We consider the creation of the general purpose arbiter delegating M resources to N clients. Firstly, the task is solved for the case of two resources being offered to two clients and preserving capability to allow both clients accessing resources simultaneously. Some ideas are proposed about how it can be modified to support more clients and more resources, as well as it is shown, how this arbiter can be simplified to become the multi-token arbiter with "migrating" token source initially presented in[5].**

I. INTRODUCTION

From a general point of view, an arbiter manages clients accessing one or several resources (Fig. 1). The communication with an arbiter is done via two-way communication channels. A channel is typically formed of the request/acknowledgement pairs of signals that can use either two- or four-phase communication.

Regardless of the protocol used, 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.[1]

Sometimes the request part of both resource and client channels can provide additional information, which can be utilised by the arbiter and 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.

All clients accessing the same resource can be completely asynchronous and unrelated to each other. This means, that requests can arrive at any time, possibly simultaneously, which would cause the metastability problems in non-arbitrated circuits.

There is a number of arbitration phenomena 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 well known E. Dijkstra's semaphores. A semaphore is the shared variable, that can be changed by many processes. The code changing this value must reside inside the critical section part, so that only one process can execute at a time. This is the example of many-to-one arbiter, because many processes (clients) access the same block of code (resource) and not more than one process can actually execute it at a time.

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 rights hand which is shared with his left and right neighbours. When all philosophers start eating by taking the forks into their right hands, 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.

[1]In simple arbiters, the resource is considered to be always available when there are no other requests from other clients. These arbiters do not need explicit resource channels

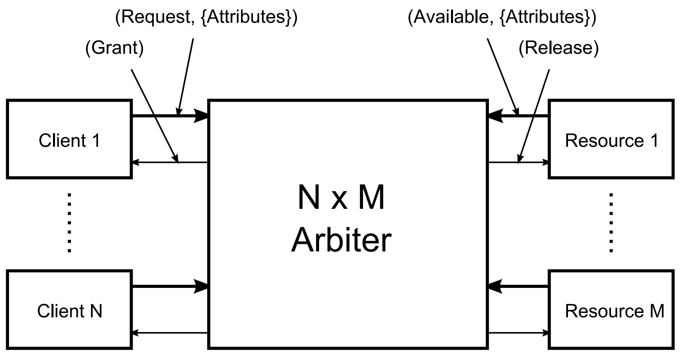

Figure 1. Synopsis of an arbiter (picture reproduced from [1])

One life example could be the elevator lifting not more than a certain amount of people. This type of resource is capable 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 grant is taking away one or may be 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. In practice, it would create an artificial bottleneck for the device throughput, and help to reduce the impact of the bursty environment.

In many cases, and particularly in the area of systems-on-chip, one should think of complex resource allocation implemented in hardware. A typical example would be the implementation of a router or a switch addressing data between, say, processors and memory blocks, which has a number of input channels (clients) and output channels (resources). This general case arbiter is considered in this paper as an attempt to solve existing problems.

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 simple 2×2 case.

A. Functionality

The arbiter is supposed to provide handshakes between East and West neighbouring circuits. Both clients and resources

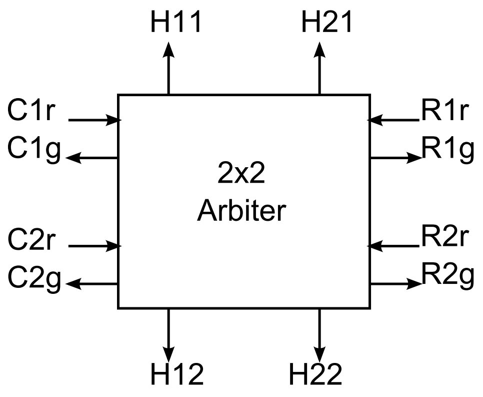

Figure 2. 2x2 arbiter interface

actively participate by producing requests (clients request, when they need a resource and resources make requests to inform the arbiter about their availability). It looks as if the clients would be searching for a resource and the resources searching for a client (Fig. 2).

When at least one resource is available and one client is requesting it, the handshake is possible. It results in an arbiter activating 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. At this point, the client (or the resource) can signal the arbiter that the job is done by removing its request. Removing it 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. It means, that the processing is finished on both sides and the channel is no longer used. For the correct functioning of the circuit, both sides have to wait until the grant is released, before beginning the preparations for the next request.

As it can be seen from the Petri net STG (Fig. 3), the channel activation has a behaviour of the mutual exclusiveness. When, say, client $C1$ is preoccupied 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 is not preventing some simultaneous handshakes. When all four requests come at the same time, arbiter makes a decision and connects requests by activating non-conflicting channels (which are $H11$ and $H22$ forming parallel handshakes or $H12$ and $H21$ forming the over-crossing handshakes).

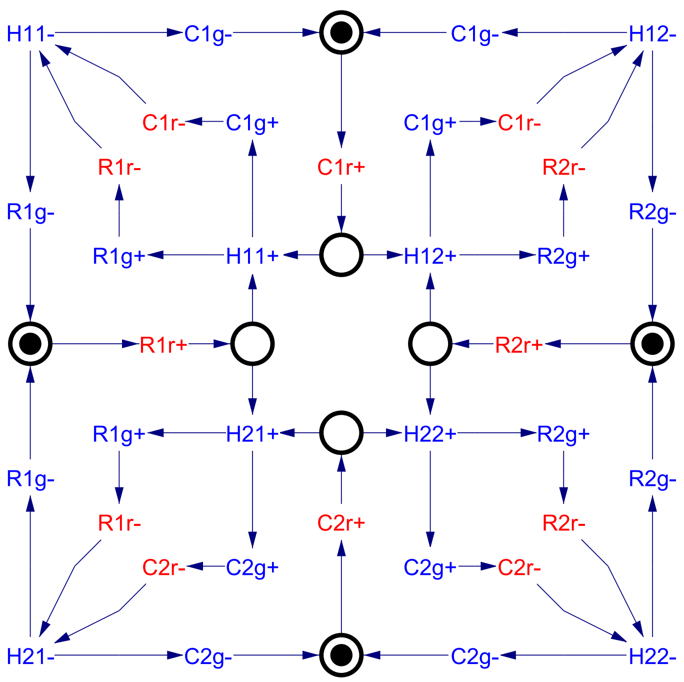

Figure 3. 2x2 arbiter STG

B. Implementation

The arbiter is constructed of several standard asynchronous elements such as C-elements, and MUTEX-es.

1) C-element: The basic, two input C-element introduced by Muller, has a behaviour described by formula:

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

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 some three input C-elements:

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

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

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

where all four signals are used to activate the output and only two deactivating it.

2) MUTEX element: MUTEX is the basic two input arbiter[2], [4], 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). In case of inputs arriving at the same time, it hides the metastability and waits until one of the signals eventually resolves it.

3) Arbiter implementation: The system is composed of several blocks responsible for particular functionality. Those are: the request mask, the request controller and the grant controller (Fig. 4).

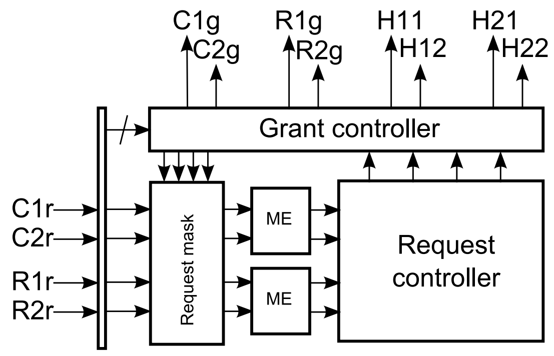

Figure 4. Arbiter structure

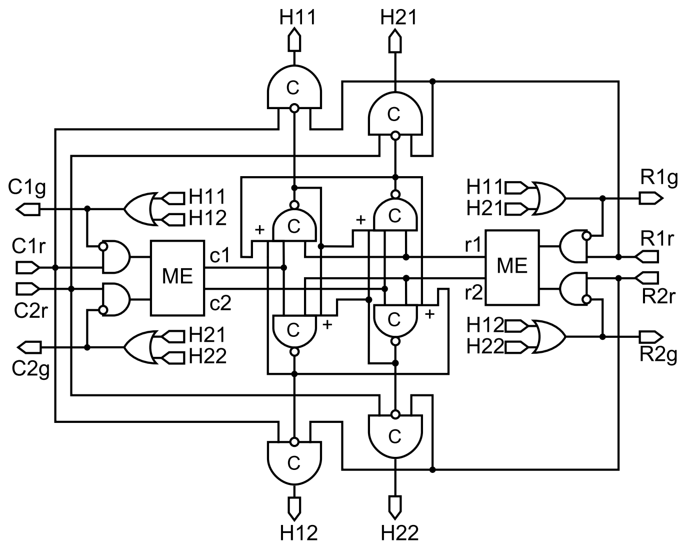

Figure 5. Arbiter Implementation

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 MUTEX (Fig. 5). Both MUTEX elements will provide that there are at most one client and one resource entering the request controller part. This effectively eliminates the choice that is always possible when three or four requests arrive at the same time. This behaviour reminds of an arbiter with enabling and eager arbitration[5], only here the enabling signal can be any request activating a handshake.

Suppose, at some point signals $C1r$ and $R1r$ win the arbitration. As soon as two arbitrated requests $c1$ and $r1$ 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, simultaneous 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

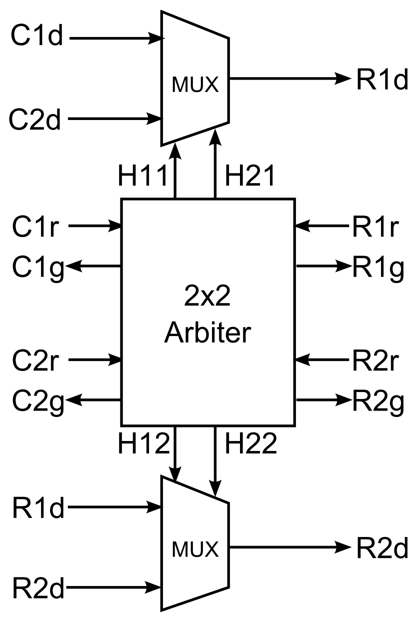

Figure 6. Arbiter usage example

$c2$ and $r2$ 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 $H22$).

4) 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 the explicit delay assumptions, request masks activated by the same channel grant can react differently and free left and right MUTEX-es at distinct moments of time. As soon as either of the MUTEX elements is freed, it lets the new request to 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 $r1$ can not be used for other handshakes. This hazard is prevented using additional inputs on the set phase of C-elements.

5) Grant controller: The grant controller consists of three-input C-gates. Each channel is activated by synchronising the requests from the associated participants and the selection of the request controller. The channel signals can be used to select the right data propagation path as shown in figure6, which is then followed by outgoing grant signal activating the access.

C. Verification of the circuit

The circuit was formally verified using Workcraft development tool, as it is described in [3]. In other words, the circuit implementation (Fig. 5) combined with the environment described in figure 3 ensures the circuit is *speed-independent* at the gate level and is not containing *hazards*.

It was also shown that there are reachable states activating simultaneous non-conflicting handshakes ($H11$ and $H22$ or

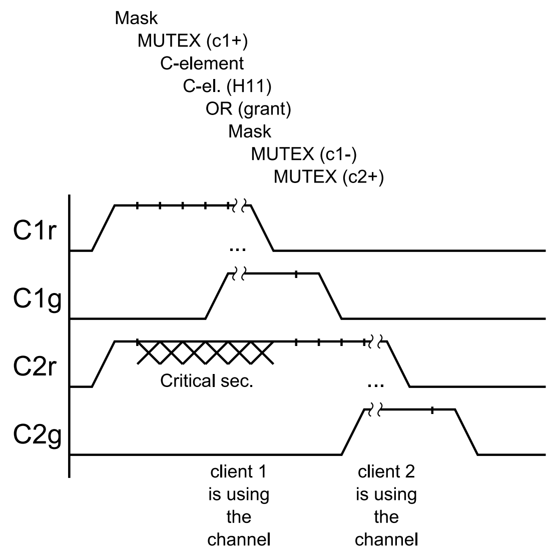

Figure 7. Timing diagram

$H12$ and $H21$) and there are no reachable states activating the rest, conflicting handshakes.

D. Latency estimation

Because the channel controller is only capable of processing one hand-shake at a time, it has a critical section delaying the grant response time to the client being 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 (Fig. 7). 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 since the moment the combination of requests actually allow the handshake.

We assume that all gate delays are the same and there is no metastability in MUTEX-es. In the beginning, both requests propagate through the request mask simultaneously and arrive into 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 if 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 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 more than the delay of critical section. It can, however, still be a safeguard if the client reaction time for $C1g+ \rightarrow C1r-$ is sufficiently long.

E. Generalising to $N \times M$ arbiter

Theoretically, it is also possible to extend the existing design to make it supporting $N \times M$ possible handshakes. To do that, we would need to use M and N input arbiters instead of MUTEX-es, that would deliver handshake candidates into the request controller. A disadvantage of such solution would be the linear growth of the worst case latency. In this case, worst delay can be estimated as:$(\min(N, M) - 1) \cdot \delta$, where δ is the critical section delay.

F. Creating Multi-Token Arbiter

Current arbiter implementation can be also easily modified to behave as the multi-token arbiter with migrating tokens presented in[5]. The migrating token arbiter consists of cells propagating tokens similarly to busy token arbiter. Tokens, however, may be removed from the chain when a client accesses the resource and then added back, when clients frees the resource. The total amount of tokens in the chain corresponds to the maximum number of clients gaining access to the resource.

If we try to imagine 2×2 arbiter as being the multi-token arbiter cell, data channel $C2d$ will be receiving tokens from previous cell, channel $C1d$ will receive tokens returned by the client, channel $R1d$ will give away tokens, when a resource is requested and, finally, channel $R2d$ will propagate the token further to the next cell. The only difference is that the proposed arbiter allows to propagate tokens from channel $C1d$ to channel $R1d$. While the multi-token arbiter does not allow situation, when a client inserting a token is trying to take it back at the same time. It means that we don't want handshake $H11$ to be activated. Disabling $H11$ can be done easily by disconnecting the $H11$ request from the grant controller.

III. Conclusions

This paper describes the asynchronous design of an arbiter managing handshakes between two clients and two resources. Each resource is actively reporting of its availability and can be connected to any of the clients. One of the features ensured is its ability to produce parallel, non-conflicting hand-shakes within reasonable time limitations. The circuit was verified to be speed-independent using automated tools.

Acknowledgements

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

References

[1] David J. Kinniment. *Synchronization and Arbitration in Digital Systems*, chapter 11 by A.Bystrov, M.Renaudin, A.Yakovlev. John Wiley & Sons, Ltd, 2007.
[2] R. C. Pearce, J. A. Field, and W. D. Little. Asynchronous arbiter module. *IEEE Trans. Comput.*, 24(9):931–932, 1975.
[3] Ivan Poliakov, Andrey Mokhov, Ashur Rafiev, Danil Sokolov, and Alex Yakovlev. Automated verification of asynchronous circuits using circuit petri nets. *Asynchronous Circuits and Systems, 2008. ASYNC '08. 14th IEEE International Symposium on*, pages 161–170, April 2008.
[4] C. L. Seitz. Ideas about arbiters. *Lambda*, 1:10–14, 1980.
[5] A. Yakovlev. Designing arbiters using petri nets. *VLSI Systems Research Center, Israel Institute of Technology, Haifa, Israel*, pages 179–201, 1995.