Generalised Asynchronous Arbiter

Stanislavs Golubcovs, Andrey Mokhov, Alex Bystrov, Danil Sokolov, Alex Yakovlev
School of Engineering, Newcastle University, UK

***Abstract*—The paper presents the design of a generalised asynchronous arbiter with a two-stage architecture that efficiently handles requests from multiple concurrent channels. The first stage of the arbiter monitors the incoming requests and locks their state as soon as one or more requests are detected. The second stage performs arbitration based on the locked state of the requests and produces the corresponding grant signals. The separation of the two stages is crucial for reducing the complexity of the arbitration logic, which allows us to obtain practical implementations for complex arbitration protocols.**

Several application examples of the generalised arbiter are proposed and evaluated in terms of scalability with respect to the growing number of request channels. The presented designs are verified to have no hazards or deadlocks using methods based on circuit Petri nets.

1. Motivation

In electronic systems arbiters are circuits that control access to one or more shared resources such as memory, various data processors or communication channels. Arbiters make discrete grant decisions based on the order and/or combination of requests from several independent sources. A basic example: two transmitters need to send data over a shared channel. They cannot use the channel simultaneously, therefore they request a dedicated arbiter to grant access to the shared resource. The arbiter in its turn guarantees that the resource is available before granting anyone access.

In synchronous systems, an arbiter works with the clocked input signals, whose grant logic is a regular synchronous automaton, and its implementation can be derived by the standard *electronic computer-aided design* (ECAD) tools. With the reducing transistor size, larger scale designs consist of multiple communicating systems (or cores) on a single chip. In such systems it becomes an increasingly complicated task to drive multiple individual cores by a single clock, which has led to the design of cores that are asynchronous with respect to each other. The communication between the cores in such systems can be done via a network of connections. Without special assumptions such a communication medium becomes a shared resource between the mutually concurrent asynchronous subsystems.

This shift from a discrete to continuous-time paradigm has a profound effect on the design of arbiters, particularly affecting their parts that compare the arrival time of input signals. It was discovered that such continuous-time decision-making circuits may produce malformed glitch pulses due to the phenomenon of metastability [1]. It was also shown that the metastability may cause unbounded delay in response of circuits [2], the effect which was generalised as a manifestation of a very old philosophical problem of "Buridan's Ass" attributed to the French philosopher of 14th century Jean Buridan. This fundamental problem of choice is formulated as the following principle: "A discrete decision based upon an input having a continuous range of values cannot be made within a bounded length of time" [3].

Since the discovery of metastability, dozens of asynchronous arbiters/synchronizers have been published [4]. The most notable designs are, probably, a cascaded serial synchronizer [5] and two-way asynchronous arbiter [6], [7], [8]. A particular case of the former is a two-flop design widely used in interfaces to synchronous circuits. It uses the clock to form a sufficiently long time interval to let the unstable meta-state to resolve itself. The latter is a set-reset asynchronous latch with an added analogue circuit suppressing propagation of the meta-state voltage to the outputs. The former is slow, as it delays propagation of signals by at least two clock cycles, and the latter has an unbounded latency, which limits its use to self-timed designs unsupported by mainstream ECAD (Electronic Computer-Aided Design) tools.

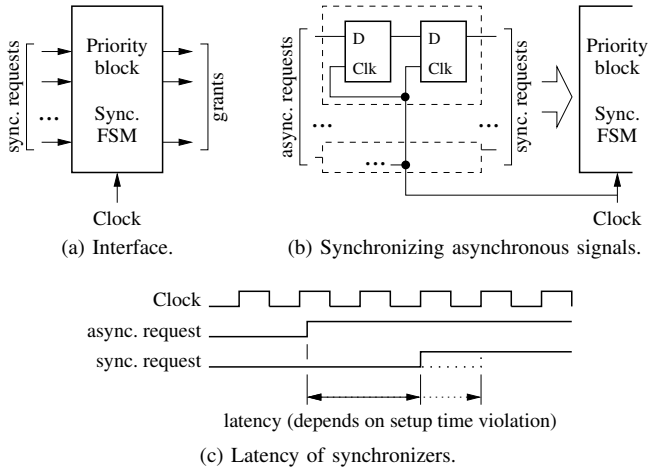
In this paper we take advantage of the low latency property of asynchronous arbiters and address the issue of lack of design flow for complex asynchronous arbitration systems adequate for modern on-chip communication structures, such as complex buses and *networks on a chip* (NoC) [9]. We use a standard architecture of a synchronous arbiter and map it into the asynchronous domain, thus enabling extremes of complexity and arriving to universal solutions. We demonstrate the flexibility of the logic design by showing the implementations of the *m*-of-*n*, priority, and "nacking" arbiters, and a general event processor.

More specifically, our contributions are as follows:

- Novel generalised arbiter with a scalable two-stage architecture and event-driven operation.
- Formal modelling and verification of the generalised arbiter using circuit Petri nets.
- Exploration of design trade-offs for latency, throughput and area by pipelining synchronisation and arbitration stages.

2. Baseline

A generic structure of an arbiter in a globally synchronous system is shown in Figure 1a. It is a standard



(a) Interface. (b) Synchronizing asynchronous signals.

(c) Latency of synchronizers.

Figure 1: Synchronous arbiter.

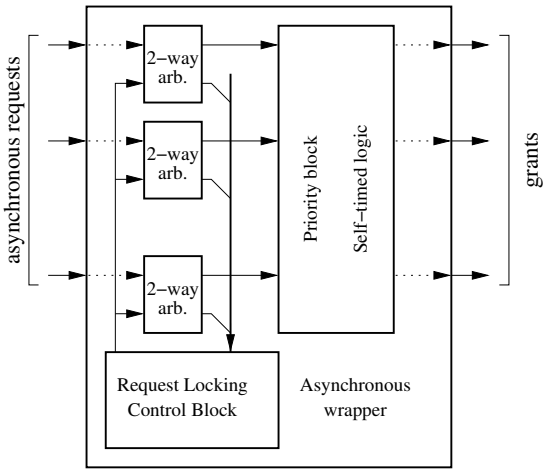


Figure 2: Asynchronous priority arbiter.

synchronous automaton which takes the bus access requests coming from the master devices on a bus and, in the next clock cycle, computes the outputs, which are grant signals. Note that such a circuit can implement any computable priority function, with an arbitrary number of requests and arbitrary number of simultaneous grants (the case of m-of-n arbitration). The complexity and scalability of the priority function are the major advantages of synchronous arbiters in comparison to the majority of existing asynchronous solutions. If a synchronous arbiter has to be used with an asynchronous bus, for example VME IEEE-1014 or any other, it needs to have its asynchronous inputs synchronised to the internal clock, as shown in Figure 1b. The synchronizers, effectively, lock the input request signals and enable its processing by standard synchronous circuits. For a slow external bus and fast internal clock it is a valid solution. However, for an SoC with fast on-chip communication fabric this solution can be viewed as a slow option, because the two-flop synchronizers add a delay of two or three clock periods to the latency, see Figure 1c.

The severity of the two-flop synchronizer latency problem is easy to illustrate. For example, for a typical $90nm$ process the metastability resolution constant τ [10], [11] is approximately $50ps$ [12], which means the time needed for reliable metastability resolution should be chosen as $T_{min} \approx 40\tau = 2ns$, thus leading to the minimum synchronizer latency of $4ns$ (two clock cycles) and minimum $6ns$ latency of the whole arbiter in Figure 1b. In our approach the two-flop synchronizers are replaced with asynchronous arbiters equipped with metastability filters [7], which do not include the worst-case delay into every synchronisation cycle, completely eliminate the risk of synchronization failures (whilst the two-flop synchronizers only reduce the failure probability to an acceptable level), and have a reasonably short average propagation delay. This leads to a significant latency reduction, as demonstrated in Section 4.4, where an asynchronous arbiter has approximately $1ns$ latency. Such a dramatic improvement, however, comes at a price. Switching to an asynchronous design methodology means difficulties with proving correctness of the design due to aspect of concurrent behaviour inherent in it. This explains this paper's emphasis on formal modelling and verification.

In the earlier work [13] an architecture of an asynchronous *priority arbiter* (PA) was developed in order to bridge the gap between the synchronous and asynchronous arbiter solutions. The PA utilises low-latency asynchronous arbiters as synchronizers clocked by a specially formed asynchronous locking signal and is capable to implement almost any priority function. The idea of separating the request capture mechanism from the priority function resembles synchronous arbiters with input synchronizers, and leads to an opportunity of implementing very complex priority functions. The architecture of PA is shown in Figure 2, where one can see a register of simple 2-way arbiters, each processing the corresponding input in a way similar to the synchronizers in Figure 1b. A disadvantage of this architecture is the use of self-timed design methodology, which at the time of publication did not support circuits with arbitration. As a result, the design examples were crafted by hand, certain timing assumptions were used, the whole design was not formally verified, and advanced priority disciplines, such as m-of-n arbitration, were not explored.

Development of two-stage asynchronous arbiters beyond the earlier published PA was held back by lack of verification tools capable to handle this class of circuits. An actively developed synthesis and verification toolkit WORKCRAFT https://workcraft.org/ supports them, thus becoming an *enabler* for a new generation of complex arbiters.

In this paper we use WORKCRAFT to verify the design of PA and the new arbiters. In particular, the architecture of the new arbiter supports *m*-of-*n* mode, "nacking" protocol, wildcards in priority vectors and a number of extensions outlined in Section 4.3. The most complex example in this paper is the "event processor" – a low-latency asynchronous device which receives data through several concurrent input channels and produces computed data on one or several outputs while ignoring any inactive inputs.

3. Generalised arbiter

3.1. Basic Structure

The basic structure of the Generalised Arbiter is inherited from the Priority Arbiter where the *grant stage* is separated from the *synchronization stage*. The choice of this structure is motivated by its good scalability because additional request rows can be added without significant latency increase. One can describe the behaviour of this arbiter with the following steps:

1) Wait until there is a change on at least one of the request lines.
2) Freeze (or lock) all input request states.
3) For the given stable set of requests, produce grants according to some desired granting logic.
4) Release locked requests and return to step 1.

A high-level circuit model of the generalised arbiter with the MUTEX elements separated from the rest of the logic is presented in Figure 3. It is a simple example of an arbiter with only two request signals $req1$ and $req2$ and two grant signals $grant1$ and $grant2$ (for clarity, we replace the number with j for referring to the j-th input channel).

Each of the request ports is restricted by the rule: $req_j = \overline{grant_j}$, which is the 4-phase communication handshake for the pair of signals req_j and $grant_j$. The transitions req_j+ and req_j- are used by the environment to request and release a resource. The $grant_j+$ signifies when the resource was granted and $grant_j-$ returns the handshake to the initial state and signals that the client is allowed to start the next transaction.

The basic components of the arbiter are the *input channels*, the $LOCKER$, and the *grant controller* (Figure 3). The input channels are used to store the state of each request and synchronize these internal states with the environment requests. The $LOCKER$ component starts synchronization when at least one of the input signals has changed its $DATA_j$ state. The grant controller is then activated to resolve the arbitration conflicts based on the request information provided by the input channels.

3.2. Process of Arbitration

Consider client requests propagating through the design. Initially, all of the circuit signals are low. The XOR gates detect any changes on the input request signals and trigger the new arbitration transaction by activating the $ME_j.r$ lines.

Suppose the basic 1-of-2 arbitration is being implemented and the $req1+$ was issued. This request propagates through the MUTEX element: $req1+ \rightarrow XOR1+ \rightarrow ME1.w+ \rightarrow LOCKER.lock+$. At this point, the $LOCKER$ component starts synchronization by locking all of the unlocked MUTEX components: $LOCKER.lock+ \rightarrow ME2.l+$. Then the internal request state is updated on the $DATA1$ component. The $DATA1$ component will align its output $DATA1.r$ with the current value of $req1$ whenever both $DATA1.lock$ and $DATA1.w$ are active:

$$DATA1.r = \begin{cases} \uparrow & req \cdot w \cdot lock \\ \downarrow & \overline{req} \cdot w \cdot lock \end{cases} \quad (1)$$

After the alignment phase is complete, the XOR gate inputs become equal and it hides the request from the MUTEX component, consequently letting the $ME1.l+$ transition to take place: $LOCKER.lock+ \rightarrow DATA1.r+ \rightarrow XOR1- \rightarrow ME1.w- \rightarrow ME1.l+$.

For any number of input channels, the special state with all the MUTEX elements holding their $ME_j.l$ signal active will signify the end of synchronization. At this point, the input signals $r1$, $r2$, and $comp$ in the grant controller GC will form a *bundled data channel* [14], where $comp$ is the request line, and $r1$, $r2$ are the data lines.

When $GC.comp+$ is triggered, the grant controller is allowed to change the grant signals based on the data from the input channels.

The design of the grant controller can be created by modelling a *finite state machine* (FSM) that updates its state whenever the $comp$ input signal is raised. After all of the grant signals have settled and are presenting the new FSM state, the grant controller acknowledges with $done+$, which eventually resets $LOCKER.lock$. This finishes the current arbitration transaction, eventually leading to $GC.comp-$ and $GC.done-$.

The important thing to note is that both requests $req1+$ and $req2+$ may arrive simultaneously and both may propagate as "win" through the column of MUTEX elements. In this case, the grant controller will process both requests within the same transaction and release grants according to its implementation.

Another important situation is when new requests arrive after the arbitration was started. These requests will be blocked by MUTEX elements until the current transaction is finished. However, once MUTEXes are released, all of the pending requests will manage to propagate to the next arbitration round.

According to the arbiter protocol, the release of a resource is eventually followed on the client side: $req1+ \rightarrow grant1+ \rightarrow req1-$. The $XOR1$ element will see this transition as a request for new arbitration, eventually igniting $ME1.w+$. This allows implementing arbiters with arbitration conflicts at the resource release phase, e.g., the "Nacking arbiter" [15].

3.3. Decompositions

This section demonstrates two scalable decompositions into simple logic gates that are commonly supported by standard cell libraries (such as TSMC). The first decomposition is a conservative design implementing "broad" bundled data communication with the grant controller module and using a pair of D-latches to control request propagation (Figure 4a). Here the signals $comp$ and $done$ are the bundled data

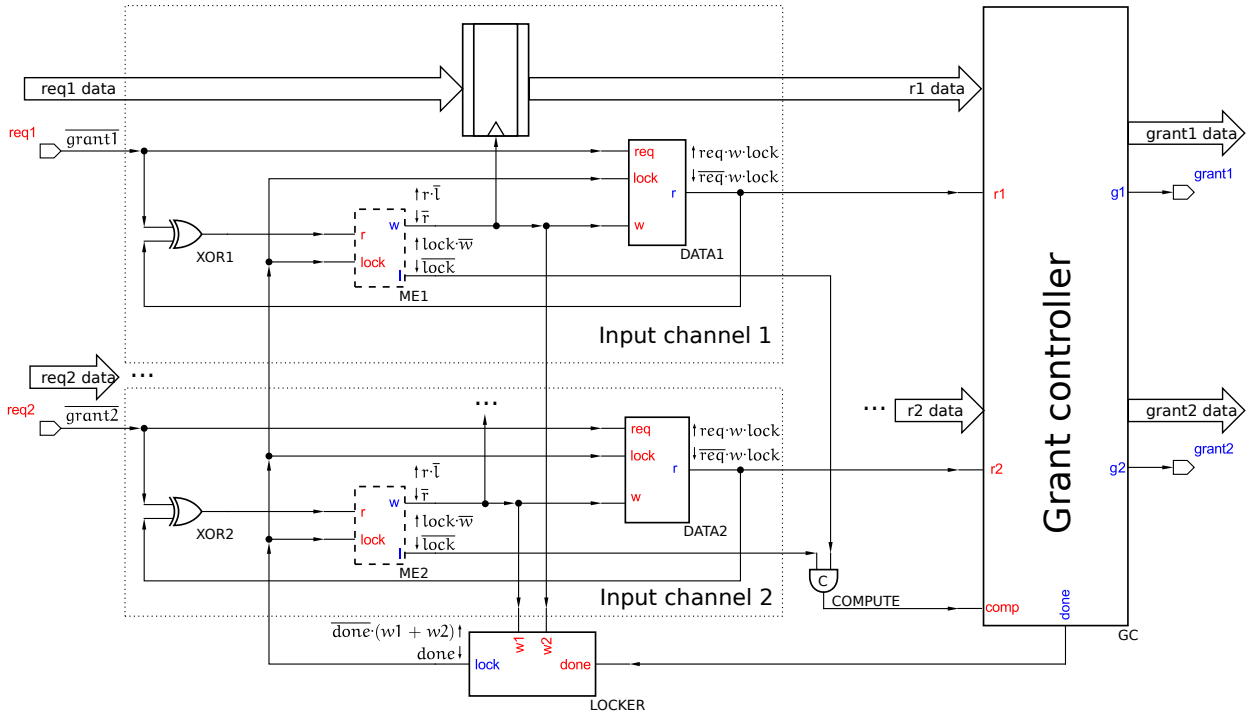
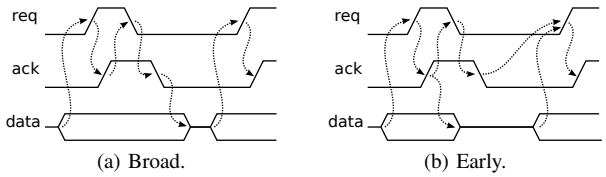


Figure 3: High-level arbiter structure.



(a) Broad. (b) Early.

Figure 4: Bundled data protocols.

handshake, and the signals $r_1, \ldots, r_n$ form the data channel wires. The second decomposition is optimised for higher throughput by concurrent release of the req and the $data$ channel, as in "early" bundled data protocol (Figure 4b).

In broad data protocol, the data channel is bound to be valid until the moment the party receiving the data releases the acknowledgement. In the early bundled data, the data channel is allowed to be reset right after the receiver has acknowledged the initial request. So, the reset phase of the four-phase handshake happens concurrently with setting up data for the next communication, which makes the protocol faster.

Broad bundled data decomposition. The broad bundled data decomposition is shown in Figure 5. The j-th channel has the signal $LLOCKj$ (the "local lock" signal) which connects to the $SYNC\_OR$ element. $LLOCKj$ is a reset-dominant latch. It is activated by $ME_j.w$ and holds active until being reset by $GC.done+$.

The signal $DATAj$ is implemented with the transparent D-latch ($DATA_j$ is transparent when $LLOCK_j$ is active). The second D-latch ENV_j is needed in case the environment reacts faster than the grant controller finishes its computation. So, while the grant controller is in the state: $GC.comp = 1$ and $GC.done = 0$, changes on the input channels that have triggered current computation are blocked (the latch ENV_j is only transparent when $LLOCK_j$ is not active).

The output signal $LCOMP_j$ ("local compute") signifies that the j-th input channel has finished synchronization and $DATA_j$ is settled to a stable value. If $ME_j.l$ is active, corresponding MUTEX element prevents the transition $ME_j.w+$, which is required to make $DATA_j$ transparent and change its value.

Early bundled data decomposition. The input channel supporting early bundle data is presented in Figure 6. The signal $LLOCK_j$ is now implemented as a set-dominant latch. It allows new requests propagating right after $ME_j.l$ is released. Hence, the reset phase $ME_j.l- \rightarrow LCOMP_j- \rightarrow COMPUTE \rightarrow GC.done-$ is concurrent with the $SYNC\_OR$'s setup for the next communication: $ME_j.l- \rightarrow ME_j.w+ \rightarrow SYNC\_OR+$. As soon as the $SYNC\_OR$ is set and the $GC.done$ is reset, the $LOCKER$'s C-element immediately starts new transaction.

4. Arbitration strategies

One possible structure for the grant controller is shown in Figure 7. It consists of the combinational logic block AP (Arbitration policies), which computes the next state of grant signals based on the current value of grants

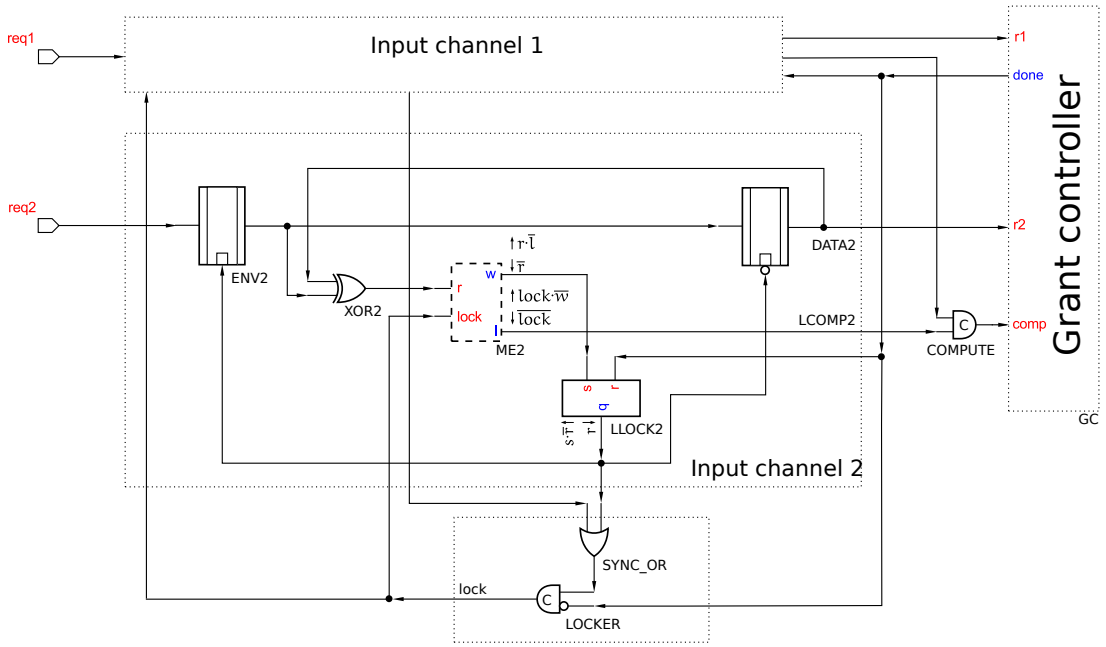


Figure 5: Broad bundled data decomposition into simple gates.

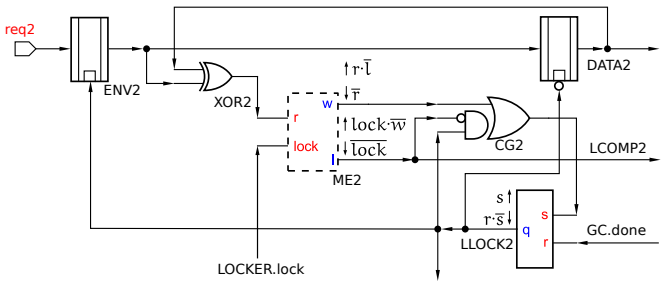


Figure 6: Early BD input channel.

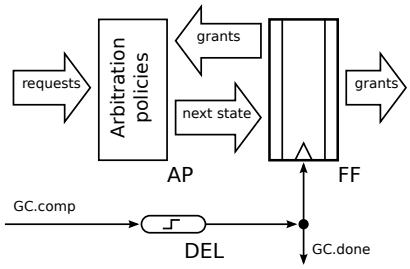


Figure 7: Grant controller structure.

$(GC.grant1, \dots)$ and the current state of the request lines (the signals $GC.r1, \dots$).

The positive edge delay element DEL receives the signal $GC.comp$. The delay time is matched with the AP module so that the "next state" values turn stable before the flip-flops FF are clocked by $GC.done$.

4.1. Avoiding Deadlocks

The arbiter does not have deadlock states if the grant controller is implemented correctly. By design, the request signals are causally related to the state of the grant signals by obeying the 4-phase handshake protocol rules. This means that each of the request states is only expected to change, when its current value is equal to the value of the corresponding grant signal.

Consider the implementation of a simple 2-input priority arbiter. When the grant controller receives a request from the second client $req2+$, it is consequently granted with $grant2+$. While the resource is being used, the first client may also initiate its request: $req1+$, which triggers the next arbitration but is not granted because the resource is busy. Eventually, the resource is released with $req2-$. The sequential grant computation logic may have a flaw that will result in ignoring $grant1+$ (because the resource is still busy). At the same time, the arbiter might release the resource with $grant2-$.

The arbiter now knows that the resource was released; however, it still needs new transitions on the request lines to begin the new arbitration. Because the first client has already sent its request, it will be waiting for a response, and the arbiter will be stalled until the next activity on the $req2$ line. Therefore, to avoid stalling, the grant controller must always provide all of the grant signals that became possible because of the released resources within the same arbitration translation. A safe way to implement the grant controller would be to first release all the resources, and only then compute client requests that will be granted.

4.2. Verification and Analysis

Analysis and verification of the asynchronous circuits is automated in WORKCRAFT software that employs Petri nets to unify the graph models of different abstraction levels in a single toolkit [16].

Decomposition. The Petri net model is generated from the decomposed circuit in Figure 5 as described in [17]. The grant controller is defined in a way that its outputs $GC.grant1$, $GC.grant2$ may arbitrarily align with the inputs $GC.r1$, $GC.r2$ while the condition $GC.comp{\cdot}\overline{GC.done}$ is true:

$$GC.grant1,2 = \begin{cases} \uparrow & comp \cdot \overline{done} \cdot r1,2 \\ \downarrow & comp \cdot \overline{done} \cdot \overline{r1,2} \end{cases} \quad (2)$$

In other words, any combination of grants is permitted for any combination of requests. This creates a potential deadlock state when both requests were ignored ($r1 = 1$, $r2 = 1$, leaving $grant1 = 0$, $grant2 = 0$) or when both resource releases were ignored ($r1 = 0$, $r2 = 0$, and $grant1 = 1$, $grant2 = 1$).

The signal $GC.done$ always aligns itself with the $GC.comp$:

$$GC.done = GC.comp \quad (3)$$

Its rising edge of signal $GC.done$ can be constrained in such a way that the deadlock is avoided:

$$GC.done\uparrow: \overline{(r1 \oplus grant1) \cdot (r2 \oplus grant2)} \quad (4)$$

So, the $GC.done+$ transition is only allowed when at least one grant signal is aligned with its input request, which ensures more request activity eventually.

After applying these constraints, the automated verification phase was successful, showing no deadlocks or hazards found.

Scaling of request locking circuitry. There is no problem while $SYNC\_OR$ is a single OR gate. However, under the assumption of arbitrary gate delay, a hazard may occur when $SYNC\_OR$ is split into a tree of OR gates.

Consider a decomposition shown in Figure 8. The signals $LLOCK1$ and $LLOCK3$ may activate simultaneously, igniting the transition of the OR gates. It is sufficient for $LLOCK3$ alone to be present in order to activate the $lock$ signal followed by the computation in the grant controller. Now, assume the transition $x+$ is particularly slow and does not happen up until $GC.done+$. As the $GC.done+$ fires, the latches $LLOCK1$, $LLOCK2$, $LLOCK3$ are reset back to 0 while also disabling the unacknowledged transition $x+$. This is a hazard condition, which might lead to a wrong computation and hence must be avoided. In any practical circuit the race between transitions $x+$ and $GC.done+$ is easy to predict because $done+$ timing will be postponed by the relatively slow $COMPUTE$ signal and the grant computation logic.

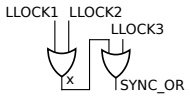


Figure 8: *SYNC_OR* decomposition.

The following timing assumption avoids this hazard:

$$GC.done\uparrow: \overline{(LLOCK1 + LLOCK2) \cdot \overline{x}} \quad (5)$$

In other words, this assumption states that every path $LLOCKj \rightarrow SYNC\_OR \rightarrow LOCKER$ has to be shorter than the path $LLOCKj \rightarrow DATAj \rightarrow MUTEXj \rightarrow COMPUTE \rightarrow GC \rightarrow LOCKER$, which in practice should be easy to ensure.

Once the timing assumption is added, the circuit can be verified to have no deadlocks or hazards. This is the only timing assumption, that is required for the circuit to work, otherwise it is not sensitive to arbitrarily slow gates and can be considered *speed-independent* (SI).

Arbiter Scaling. Arbiter scaling up to N clients is straightforward because each of the input channels is a *true tile* [18]. Each true tile has a fixed communication interface that does not change with the increased number of clients.

On contrary, with the increasing number of input channels, the $SYNC\_OR$ and the $COMPUTE$ components (Figure 5) have to increase the number of inputs, one input per each input channel. The C-element can be safely composed as a tree of C-elements (or also it can be built with AND and OR gate trees followed by a C-element). The $SYNC\_OR$ is formed as a tree of OR gates. Decomposing it into multiple OR gates requires adding a timing assumption, which was described later in section about circuit verification.

Latency. Because of significant synchronization overhead, the arbitration latency may seem large for the occasional requests arriving scattered over time. However, the arbiter latency is expected not to change much when different numbers or requests are processed. While the arbiter is occupied processing one request, several scattered requests may accumulate and become pending. The high performance of the arbiter is achieved by the fact that all of the pending requests are processed in a single arbitration transaction. Hence, providing there are enough resources, the arbiter issues grants concurrently to all clients.

The design of the grant controller is built within the bundled data interface. This means that the grant controller can be developed from a finite state machine specification, where rising edge of the signal $GC.comp$ activates FSM state transition. At the same time this transition is only activated when there is a candidate for computation, meaning that the dynamic power consumption is minimised.

4.3. Possible Extensions

The flexibility of the generalised arbiter allows implementing various extensions. Some of them are suggested in

this section.

Data Lines. The request lines can be bundled with data lines to provide additional information for the arbitration logic. One practical example would be implementing the priority-based arbitration with dynamic priorities. The arbitration policy could use the data lines to distinguish, which of the requests is more important during current arbitration transaction. Alternatively, the data lines can be used to specify, which service does the client need. With this information, the arbiter may grant a compatible resource from a resource pool as soon as it becomes available.

The data lines do not need to be arbitrated because they are aligned with request lines. However, they would still need to be latched by the signal $ME_j.w$ so that the arbitration logic always works with stable input data.

Accumulate and Fire. The *accumulate and fire* [19] tactics can be enforced by modifying the OR-gate tree in the $LOCKER$ component in order to ignore the request combinations that are not useful or interesting. For instance, the arbiter may wait for at least a few requests arriving before it actually starts the arbitration and does not waste energy on lonely requests. This is also the case for the $M \times N$ arbiter [20] that matches request pairs. It is impossible to match a pair with only one request active, hence,no need to begin the arbitration.

Another important aspect for accumulating requests is that the grant controller may never receive certain combination of inputs, as a result, there will be additional "don't care" states, which may significantly simplify it's logic.

Pipelining. *Pipelining* is a technique used in long wire interconnects to increase the throughput of a system [21], [22]. It also increases throughput by splitting slow computation into multiple fast computation stages. Similar approaches can be used to improve the throughput of the arbiter by splitting its synchronization and grant phases into separate pipeline stages. The first stage only fulfils the synchronization and forwards a set of stable requests to the next stage through a handshake. As soon as the data is passed on, the new synchronization may begin. The next pipeline stage only computes grants signals based on the request state received and has a higher throughput as a result.

Because the grant phase latency will be different for the different versions of the arbiter, we use such a design to estimate the performance of the synchronization phase without the grant phase.

4.4. Application Examples

Implementing different arbitration policies is a straightforward task:

- Identify the rules that have to be followed and create corresponding truth table.
- Synthesise circuit with any logic synthesis tool. In our examples we used the LOGIC FRIDAY frontend to ESPRESSO logic minimiser [23].

TABLE 1: Priority 1-of-3 arbiter

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g1'$	$g2'$	$g3'$
1	X	X	0	X	0	1		
1	X	X	X	0	1	1		
1	X	0	X	X	0	1		
X	X	1	1	X	X		1	
0	X	1	X	0	X		1	
0	X	1	X	X	0		1	
X	X	X	X	1	1			1
0	X	0	X	1	X			1
other combinations						0	0	0

TABLE 2: 1-of-3 with rotating priority

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g1'$	$g2'$	$g3'$
1	1	X	X	X	X	1		
1	X	0	X	0	X	1		
1	X	1	0	X	0	1		
1	X	X	X	0	1	1		
X	X	1	1	X	X		1	
0	X	1	X	0	X		1	
0	X	1	X	X	0		1	
X	X	X	X	1	1			1
0	X	0	X	1	X			1
X	0	0	X	1	X			1
other combinations						0	0	0

Priority 1-of-3 Arbitration. The first example demonstrates the 1-of-3 priority arbitration. Here $g1'$, $g2'$, and $g3'$ are the next state values for the current state of requests $r1, \ldots, r3$, and grants $g1, \ldots, g3$.

Any requesting client gets a resource as long as no other client is being currently granted. The second client only gets a resource only if the first is not requesting and the third is not being granted. The third client only gets a resource if no other client is requesting it. Thus we have a simple chain (linear order) of priority. The generated truth table is shown in Table 1.

The equations implementing the circuit are as follows:

$$\begin{aligned} g1' &= r1 \cdot (\overline{g3} \cdot (\overline{g2} + \overline{r2}) + \overline{r3} \cdot g3) \\ g2' &= r2 \cdot (\overline{r1} \cdot (\overline{g3} + \overline{r3}) + g2) \\ g3' &= r3 \cdot (\overline{r1} \cdot \overline{r2} + g3) \end{aligned} \tag{6}$$

Note, that in this arbiter the third channel may starve, i.e. never receive the grant signal, if each of the lower two channels produces a new request immediately after the corresponding grant is withdrawn. It is a good illustration of the effect of arbitration priority on system properties. A known method of avoiding starvation of clients is to "rotate" the priorities, thus giving the highest priority to every client at some point. Our generalised arbiter can implement such a priority discipline, where $r1$ has a higher priority than $r2$, $r2$ is higher than $r3$, and $r3$ is higher than $r1$. So, when a resource is released, it is immediately granted to the next client in the line, and the line is looped back as a circle. If all three requests arrive simultaneously, then the $g1$ is issued first.

$$\begin{aligned} g1' &= r1 \cdot (r2 \cdot \overline{g2} \cdot \overline{g3} + \overline{r3} \cdot (g3 + \overline{r2}) + g1) \\ g2' &= r2 \cdot (\overline{r1} \cdot (\overline{g3} + \overline{r3}) + g2) \\ g3' &= r3 \cdot (\overline{r2} \cdot (\overline{g1} + \overline{r1}) + g3) \end{aligned} \quad (7)$$

An example of specification of the priority logic function is shown in Table 2 and equations (7). Please note that not all input combinations of this function are used in the arbiter; for example 111111 would have meant that all three channels were in the process of granting the requests simultaneously, which is not a reachable state of the system.

This arbiter was implemented with bundled-data priority logic block. The latency (request to grant time, if the grant is not blocked by the other channels) based on a CMOS 90nm implementation (including wire delays) of a request propagating from the input port until receiving a grant signal is between 900ps and 1000ps (depending on the request vector). This is a significantly shorter latency than $6ns$ in the synchronous design with two-flop synchronisers at the request inputs shown in Figure 1b and discussed in Section 2.

Priority 2-of-3 Arbitration. Another example is a 2-of-3 arbiter. It acts similarly to the 1-of-3 priority arbiter, favouring requests $r1$, $r2$, $r3$ in that order. However, it is allowed to grant 2 resources at a time. This example demonstrates implementing multi-resource arbitration (see Table 3).

It is implementable with the following equations:

$$\begin{aligned} g1' &= r1 \cdot \overline{r2 \cdot g2 \cdot r3 \cdot g3} \\ g2' &= r2 \cdot (\overline{r1 \cdot g3 \cdot r3} + g2) \\ g3' &= r3 \cdot (\overline{r1 \cdot r2} + g3) \end{aligned} \quad (8)$$

Nacking arbiter. A Nacking arbiter [24], [15] has two types of acknowledgements. In each arbitration cycle it either grants a resource or responds with "not granted" signal indicating that the resource is not available. It allows designers to build systems, where a client does not become locked in a pending state while being "informed" that it can

TABLE 3: Priority 2-of-3 arbiter

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g1'$
1	X	0	X	X	X	1
1	X	X	0	X	X	1
1	X	X	X	0	X	1
1	X	X	X	X	0	1
other combinations						0

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g2'$
0	X	1	X	X	X	1
X	X	1	1	X	X	1
X	X	1	X	0	X	1
X	X	1	X	X	0	1
other combinations						0

$r1$	$g1$	$r2$	$g2$	$r3$	$g3$	$g3'$
0	X	X	X	1	X	1
X	X	0	X	1	X	1
X	X	X	X	1	1	1
other combinations						0

TABLE 4: Nacking arbiter

$r1$	$g1$	$n1$	$r2$	$g2$	$n2$	$g1'$	$n1'$	$g2'$	$n2'$
1	X	X	0	X	X	1			
1	X	X	X	0	X	1			
1	X	X	1	1	X		1		
0	X	X	1	X	X			1	
X	X	X	1	1	X			1	
1	X	X	1	0	X				1
other combinations						0	0	0	0

do something else in this time. An example of the priority function of a two-way nacking arbiter is constructed in Table 4 and equations (9). Again, not all input combinations of the logic function are reachable due to constraints imposed by the protocol of operation.

$$\begin{aligned} g1' &= r1 \cdot \overline{r2 \cdot g2} \\ n1' &= r1 \cdot r2 \cdot g2 \\ g2' &= r2 \cdot (\overline{r1} + g2) \\ n2' &= r1 \cdot r2 \cdot \overline{g2} \end{aligned} \quad (9)$$

Event processor. The generalised arbiter architecture can be extended further by adding data buses to each of request inputs; the same idea as the *dynamic priority arbiter* (DPA) described in [13]. The beauty of this architecture is that it only processes the data inputs selected by their respective request signals, while ignoring the others. In the asynchronous domain it is a non-trivial problem, because of the possible race conditions on the inputs. The input data buses can carry information about the priority of a request, thus implementing the example from [13].

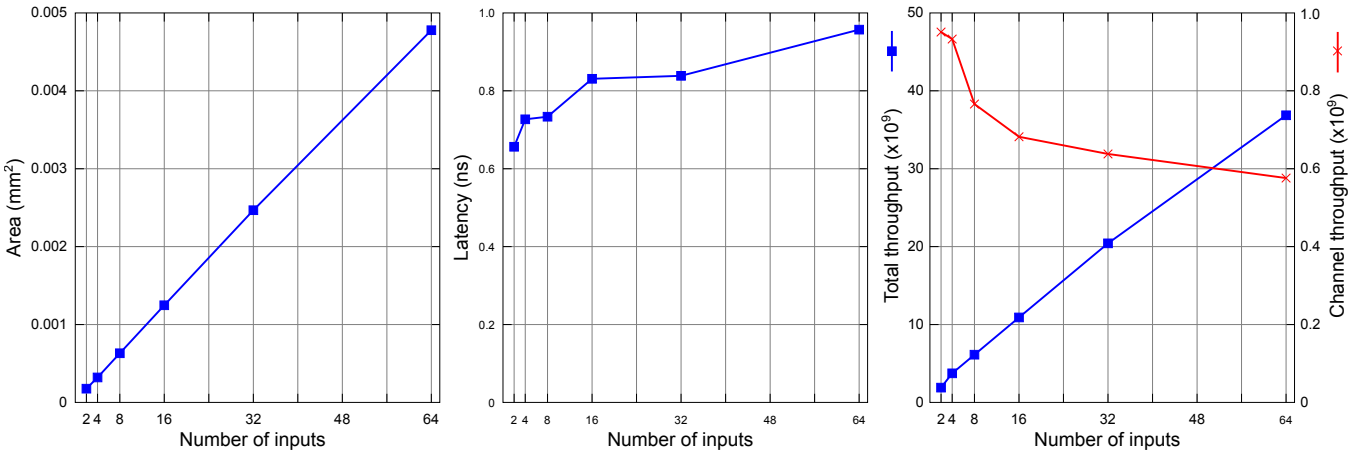
We develop further the idea of DPA in this paper by adding data buses to each grant output, and call it an event processor. An event processor receives data through independent asynchronous channels, takes care of race conditions and generates output data. It is important the request-grant pairs form handshake interfaces for each input channel; while selection of the crossbar output is represented as data associated with the grants. For example, the input channel $(req1, req1\_data)$ may request the crossbar output #2 by setting $req1\_data = 2$; then the resource will be granted by returning the signal on $grant1$ and setting $grant1\_data = 2$.

One may see that such an event processor is essentially the earlier described generalised arbiter with added data channels associated with each request and grant. Request-grant pairs form handshakes controlling the corresponding data channels. In our implementation we use a bundled data approach. If compared to the earlier idea of DPA, this design has additional output data channels, and its protocol is extended to process the falling transitions on request lines.

In the following example we demonstrate the power of the event processor architecture by implementing a decision-making element of a NoC router node. For simplicity assume that the node is 2x2 crossbar, where each of two input channels may request any of the output two channels. If both inputs request the same channel, only one request will be granted. However, if the requesters aim at different

Priority logic inputs								Priority logic outputs				Cond.	Comment	
r_1	w_1	g_1	d_1	r_2	w_2	g_2	d_2	g_1'	d_1'	g_2'	d_2'			
0				0				0		0			both requests removed	
0				1	a	0		0		1	$sel(a)$		r_2 added	one req.
0				1		1	a	0		1	a		r_1 removed	
1	a	0		0				1	$sel(a)$	0			r_1 added	
1		1	a	0				1	a	0			r_2 removed	
1	01	0		1	01	0		1	01	0			conflict on resource	two new req-s
1	01	0		1	10	0		1	01	1	10		mutually exclusive wildcards	
1	10	0		1	01	0		1	10	1	01		mutually exclusive wildcards	
1	10	0		1	10	0		0		1	10		conflict on resource	
1	11	0		1	01	0		1	10	1	01		one broad wildcard	
1	11	0		1	10	0		1	01	1	10		one broad wildcard	
1	01	0		1	11	0		1	01	1	10		one broad wildcard	
1	10	0		1	11	0		1	10	1	01		one broad wildcard	
1	11	0		1	11	0		1	$sel(11)$, e.g.10	1	$11\backslash d_1'$, e.g. 01		both wildcards are broad	
1		1	a	1	b	0		1	a	0		$a = b$	conflict on resource	new & old
1		1	a	1	b	0		1	a	1	$b\backslash a$	$a \neq b$	one broad wildcard	
1	a	0		1		1	b	0		1	b	$a = b$	conflict on resource	
1	a	0		1		1	b	1	$a\backslash b$	1	b	$a \neq b$	one broad wildcard	

TABLE 5: Event processor for a 2x2 crossbar controller.



(a) Area. (b) Latency. (c) Throughput.

Figure 9: Pipelined arbiter performance.

outputs, then both will be connected simultaneously. To make the design problem more interesting, the request data may include wildcards, i.e. a channel may request any output channel rather than a particular one. The logic function of the grant controller is constructed in Table 5.

This table includes request signals r_i arriving together with the corresponding wildcards w_i, which are two wire buses, where each bit is allocated on the corresponding destination. The outputs are grant signals g_i' corresponding to the requesting inputs, as in the previous designs. The grants are accompanied with the allocated destination output data d_i' buses, which are the resolved corresponding wildcards. As before, the apostrophe symbol means that these are the next-state values. The previously generated current values of the output are used in the calculation of the next state; they are treated as inputs and denoted as g_i and d_i. Several entries in the truth table are conditional - this notation is used to shorten the representation; many rows corresponding to unreachable states of the system are not defined.

5. Performance Estimations

The estimated latency of a 2-of-3 priority arbiter is based on a CMOS 90nm implementation (including wire delays) of a request propagating from the input port until receiving a grant signal is between $900ps$ and $1000ps$ (depending on which requests were issued). The latency is still roughly the same regardless of whether two requests are granted concurrently or some request is granted while another released.

Analysis of the pipelined arbiter with a variable number of input channels is detailed in Figure 9. All the measurements are for the circuits implemented in 90nm technology. As one would expect, the circuit area is linearly proportional to the number of input channels, see Figure 9a.

The latency of the pipelined arbiter for a variable number of inputs gives an idea on how fast the synchronization phase is without the actual grant computation. As it would

be expected from the tree-like structure computing the $GC.comp$ signal, the latency increase is logarithmic with the increased number of inputs. The additional input channels occasionally add more layers to the trees of logic gates communicating the input channels, thus slightly deviating from the expected latency, see Figure 9b. The latency for two inputs is $650ps$ on average. As the number of inputs increases up until 64, the latency increases to approximately $950ps$. This is still considerably faster than the $4ns$ latency of a clocked synchronizer in Section 2.

The throughput is measured as the number of input channels multiplied by the number of transactions (measured in millions per second) assuming that the grant controller is fast and the synchronizer is the bottleneck of performance. With the increased input count the total number of arbitrations drops to about 576 million arbitrations per second while the maximum number of requests processed increases to $576 \times 64 = 36864$ million requests per second, see Figure 9c.

6. Conclusions

The paper presents design of a new generalised arbiter, which has a canonical architecture able to tackle a large variety of arbitration problems. The generalised arbiter scales well having logarithmic latency increase for the increased number of inputs. All of the necessary timing assumptions used are practically implementable even without dedicated delay elements. The arbiter can support any priority discipline expressed as a combinational logic circuit. All of the arbitration transactions are atomic, thus eliminating any deadlocks that may occur in other examples with distributed arbitration (such as the "Dining Philosophers" problem).

The grant controller is activated by the locally generated signal similar to clock in synchronous designs. This allows use of standard ECAD tools for building the priority logic. At the same time, the arbiter operates only when there is a need to do arbitration, meaning that no dynamic power is wasted when input signals are idle. The arbiter also allows pipelining, which splits synchronization and arbitration into separate pipeline stages and improves the overall throughput. In combination these properties enable flexibility in choosing among various designs favouring either the smaller latency or the reduced area on dice.

The generalised arbiter has been formally verified to be free from deadlocks and hazards. However, the possibility of stalling such an arbiter may still depend on the correctness of the priority function. Further research is needed to estimate arbiter performance in various practical applications. It is believed that such an arbiter can be useful in the asynchronous NoC routers [9] and the asynchronous schedulers, where sophisticated arbitration logic is needed for designing fast, reliable and adaptive circuits operating in busy asynchronous request environments.

Acknowledgements. EPSRC supported this work by grant EP/N023641/1 *STRATA; Layers for Structuring Trustworthy Ambient Systems* and IAA grant *Waveform-based design flow for A4A circuits*.

References

[1] T. Chaney, S. Ornstein, and W. Littlefield, "Beware the synchronizer," in *IEEE Compcon*, 1972.

[2] M. Pechoucek, "Anomalous response times of input synchronizers," *IEEE Transactions on Computers*, vol. C-25, pp. 133–139, 1976.

[3] L. Lamport, "Buridan's principle," *Foundations of Physics*, vol. 42, pp. 1056–1066, 2012.

[4] D. Kinniment, *Synchronization and arbitration in digital systems*. John Wiley and Sons, 2008.

[5] L. Kellman and A. Cantoni, "Metastable behavior in digital systems," *IEEE Design and Test of Computers*, pp. 4–19, 1987.

[6] R. Pearce, J. Field, and W. Little, "Asynchronous arbiter module," *IEEE Transactions on Computers*, vol. 24, no. 9, pp. 931–932, 1975.

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

[8] A. Martin, "On Seitz's arbiter," Tech. Rep. 5212:TR:86, Caltech Computer Science, 1986.

[9] R. Dobkin, R. Ginosar, and A. Kolodny, "QNoC asynchronous router," *Integration, VLSI Journal*, vol. 42, pp. 103–115, 2009.

[10] D. Kinniment and D. Edwards, "Circuit technology in a large computer system," in *Conference on Computers–Systems and Technology*, pp. 441–450, 1972.

[11] D. Kinniment, A. Bystrov, and A. Yakovlev, "Synchronization circuit performance," *IEEE Journal of Solid-State Circuits*, vol. 37, no. 2, 2002.

[12] S. Beer, R. Ginosar, M. Priel, R. Dobkin, and A. Kolodny, "The devolution of synchronizers," in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 94–103, 2010.

[13] A. Bystrov, D. Kinniment, and A. Yakovlev, "Priority arbiters," in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 128–137, 2000.

[14] J. Sparso and S. Furber, *Principles of asynchronous circuit design*. Kluwer Academic Publishers, 2002.

[15] J. Cortadella, L. Lavagno, P. Vanbekbergen, and A. Yakovlev, "Designing asynchronous circuits from behavioural specifications with internal conflicts," in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 106–115, 1994.

[16] D. Sokolov, V. Khomenko, and A. Mokhov, "Workcraft: Ten years later," in *This asynchronous world. Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday* (A. Mokhov, ed.), Newcastle University, 2016. Available online http://async.org.uk/ay-festschrift/paper25-Alex-Festschrift.pdf.

[17] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, and A. Yakovlev, "Automated verification of asynchronous circuits using circuit Petri nets," in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2008.

[18] D. Shang, F. Xia, S. Golubcovs, and A. Yakovlev, "The magic rule of tiles: Virtual delay insensitivity," in *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pp. 286–296, 2009.

[19] Y. Chen, *High level modelling and design of a low power event processor*. PhD thesis, Newcastle University, 2009.

[20] S. Golubcovs, D. Shang, F. Xia, A. Mokhov, and A. Yakovlev, "Modular approach to multi-resource arbiter design," in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 107–116, 2009.

[21] J. Bainbridge and S. Furber, "CHAIN: A delay-insensitive chip area interconnect," *IEEE Micro*, vol. 22, pp. 16–23, 2002.

[22] R. Ho, J. Gainsley, and R. Drost, "Long wires and asynchronous control," in *International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pp. 240–249, 2004.

[23] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel, *Logic minimization algorithms for VLSI synthesis*. Kluwer Academic Publishers, 1984.

[24] S. Nowick and D. Dill, "Practicality of state-machine verification of speed-independent circuits," pp. 266–269, 1989.