Ned Bingham, Rajit Manohar. “A Systematic Approach for Arbitration Expressions.” IEEE Transactions on Circuits and Systems I.

A Systematic Approach for Arbitration Expressions

Ned Bingham and Rajit Manohar
Computer Systems Lab
Yale University
New Haven, CT
edward.bingham@yale.edu and rajit.manohar@yale.edu

***Abstract*—Greedy arbiters and bundling merges compose simultaneous events in sequence and parallel respectively. Previous designs for these problems handle two to three inputs, and can be composed in a tree topology to handle more. In addition, they include subtle timing assumptions beyond the QDI delay model and undocumented timing assumptions in their arbiter's digital model. In this paper, we discuss two slightly different digital models that we call the ideal arbiter and buffered arbiter models, and match them to CMOS implementations. From CHP specifications of the greedy arbiter and bundling merge, we derive the Maybe Execute Element. We then show how it may be systematically composed to produce improved circuits for both which use a small number of simple gates, strictly abide by the QDI delay model, and gracefully scale to an arbitrary number of inputs. Finally, we touch on how this approach may be used to develop scalable circuit solutions to more sophisticated arbitration problems.**

Keywords—arbitration, asynchronous, quasi-delay insensitive, qdi, non-deterministic, greedy arbiter, bundling merge

I. Introduction

In asynchronous logic, the implementation of non-deterministic choice requires the use of two-input two-output mutual exclusion elements. A mutual exclusion element has the following behavior: if one of its inputs is asserted, then the corresponding output is asserted. However, if both inputs are asserted, then it asserts only one of the outputs, picking between the two *arbitrarily*. Because any circuit that implements this deceptively simple behavior can exhibit metastability, careful analog analysis is required to ensure correct operation [5][6].

These circuits may be divided into two classes: Synchronizers and Arbiters. Synchronizers handle the general case in which the asserted inputs may be unstable meaning that they may be deasserted before its corresponding output has been asserted [10]. Arbiters handle the more specific case in which the asserted inputs remain stable. While synchronizers require fewer environmental restrictions, they are also a more complex circuit. Furthermore, [17] observed that it is possible to implement unstable conditions in certain practical cases by exploiting the assumption that a CMOS arbiter is in fact a fair circuit when both of its inputs are asserted. Hence, non-deterministic choices in asynchronous circuits almost always use standard arbiters as their building blocks.

One common use for an arbiter involves implementing mutually exclusive access to a shared resource. Two processes may request access to a shared resource through a third arbitrating process commonly referred to as a merge element or mixer [4]. These elements have further been used in a variety of mutual exclusion problems. For example, [7] used the ideas from [8] to communicate pulses within a neuromorphic system.

The requesting processes, the shared resource, and the merge element all communicate across *channels*. Each channel is a pair of request/acknowledge wires on which two communicating processes execute a four phase handshake protocol. If either requesting process asserts its request, the merge element must first request a lock ensuring the availability of the shared resource. When the shared resource acknowledges that request with a grant, the merge element may dispatch that grant by acknowledging one of the input requests. Once the requesting process is done and lowers its request, the merge element unlocks the resource and the entire cycle begins again. The first half of the handshake is used to request the resource, and the second half is used to release the resource.

However, there are other scenarios that require more complex constraints for access to a shared resource. For example, two simultaneous requests could be granted sequentially before unlocking the resource as implemented by a *greedy* arbiter. This is used in [2] to optimize tree-arbitrated exclusive access of many channels to a single bus, in [3] to arbitrate data received from pixels in an event-driven image sensor, and in [7] for asynchronous address event communication in spiking neural networks.

Alternatively, both requests could be granted concurrently as implemented by a bundling merge in [1]. An example of this involves using a counter to track the number of in-flight data items in a variable latency pipeline [28]. When a data value enters the pipeline, the counter is incremented; when a value leaves the pipeline, the counter is decremented. If the increment and decrement requests occur simultaneously, a bundling merge could combine them and skip both operations.

In the context of spiking neural networks with lossy communication, a bundling merge could be used to merge multiple near-simultaneous spikes into a single spike, reducing communication cost at the expense of fidelity. If you are designing an event-driven image sensor, events from multiple pixels in a column could be resolved by a read of the whole column.

One can imagine more complex arbitration problems that come from the interaction between these two. For example, a one-writer two-readers lock could mediate access to a shared memory with a write-only port and two read-only ports. Or, the counter in [28] could have a clear signal to execute a squash across the pipeline. This would have to be mutually exclusive from the execution of the bundled increment and decrement requests. These scenarios require generalized arbitration *expressions*. While this could be implemented by composing the greedy arbiter and bundling merge in various ways, arbitration trees are an inefficient way to solve the problem for a small to medium number of inputs.

Finally, there are scenarios that cannot be implemented via composition. Circling back to the counter in [28], suppose there are two entry and two egress points to the pipeline and the throughput of increment and decrement requests is above and beyond what the counter can manage. Then, you want to bundle any two requests allowing you to either skip the least significant bit of the counter or skip the counter altogether thereby reducing the throughput requirement on the counter by a factor of two.

In this paper, we re-examine the previous implementations of circuits for the greedy arbiter [2][3] and bundling merge [1], analyze their timing assumptions, and propose alternative templates for more robust arbitration expressions. In Section 2, we discuss the digital model used to analyze the arbiter's behavior, introducing a distinction between the ideal and buffered arbiters and their CMOS implementations. Then, in Section 3, we describe how the application of this model affects previously designed arbitration circuits. Section 4 proposes a new building block, the *maybe execute* element, as the basis for a family of arbitration problems which, in Section 5, is used to construct the bundling merge and greedy arbiter. In Section 6, we evaluate our design performance and compare it against the previous designs. Finally, the Appendix gives an overview of the program and circuit notation used in this paper.

II. Digital Abstractions for Arbitration

Under the delay insensitive (DI) delay model, a circuit should operate correctly independent of gate and wire delays. Correct operation means that the circuit remains stable, non-interfering, and deadlock-free. An instability, or glitch, can cause data-loss or lead to interference; interference, or a short, can cause permanent circuit damage; and deadlock halts the computation prematurely. To achieve this goal, every transition must acknowledge every input to its driving gate. A transition `a` acknowledges another `b` if there is a causal sequence of transitions from `a` to `b` that prevents `b` from firing until after `a` has completed.[26]

In order for this model to be Turing Complete, the quasi-delay insensitive (QDI) delay model makes one exception to acknowledgement called the Isochronic Fork Assumption. If there is a wire fork to multiple gates, and one of those gates does not acknowledge all of the transitions on that wire, then we assume that the delay from the driver to the non-acknowledging gate is bounded. In this model it is always safe to place an inverter before the wire fork. However, because gates have unbounded delay, placing an inverter after an isochronic fork and before the non-acknowledging gate can cause an instability. Because the isochronic fork timing assumption is easy to guarantee and maintain, real QDI circuits are *robust by construction* to temperature variation, process variation, sizing, noise, etc [24]. For a more detailed discussion on the QDI model and this timing assumption, see [26], [16], and [19].

The standard arbiter circuit in Fig. 1 is responsible for ensuring mutually exclusive outputs given two inputs. In effect, it determines which of the two input requests arrived first. Ultimately, it is an analog circuit and must be carefully verified using analog analysis. Therefore, digital simulators must explicitly model the arbiter's behavior as a black-box. The implementations of the bundling merge in [1], and greedy arbiter in [2] along with many others in the literature assume an ideal arbiter in which the two outputs are always mutually exclusive high. This section examines the analog characteristics of the circuit used to implement an arbiter, and focuses on appropriate digital abstractions for different arbiter circuits.

The arbiter in Fig. 1 consists of two stages: an RS latch followed by an instability filter (also called a metastability filter). If both `a` and `b` transition to `Vdd` simultaneously, then the RS-latch can become metastable with both `_u` and `_v` only partially transitioning. At resolution, one of the two will finish its transition to `GND` while the other will return to `Vdd`, causing a glitch [5]. Therefore, the second stage implementing an instability filter is required that converts the glitch caused by the metastable latch into a delay. According to [13], an NMOS form was first designed by Sutherland in 1976 which later appeared in [9] and [12]. To our knowledge, the CMOS form we use first appears in [11]. This arbiter will be expressed in circuit diagrams as a box labelled "Arb".

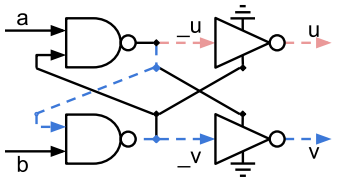

Fig. 1. Circuit diagram for the standard Arbiter design.

The inverters in the instability filter have unpredictable delay which depends on their load as determined externally to the arbiter. Because the downgoing transition of one output as highlighted blue in Fig. 1 is allowed to happen in parallel with the upgoing transition of the other as highlighted red, they can both be high at the same time, violating the blackbox model from the literature. This has practical implications, for example in the next section we analyze circuits in the literature that depend upon the outputs of the arbiter being mutually exclusive which is something that this circuit does not

guarantee.

In practice, as long as the two outputs u and v drive similar capacitive loads, then the blackbox digital model holds true and the two outputs remain mutually exclusive high. However, should they end up driving different loads, that model will fail. Suppose u is driving a high capacitive load and v is not. If a, b, and u are at Vdd and v is at GND, then lowering a could cause a transient state in which both u and v are high. A SPICE simulation in Fig. 2 replicates this behavior, showing both u and v above the threshold voltage after 0.09 ns.

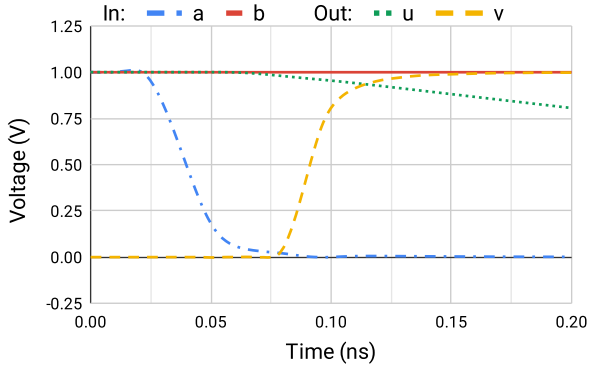

Fig. 2. SPICE waveform for the standard Arbiter design.

In effect, the blackbox model used in the literature has an extra timing assumption. As suggested by [21], we could treat this arbiter as a blackbox gate in which u and v are treated as an isochronic fork, and guarantee the output loads in layout. However, this is an assumption that is often lost when publishing circuits which use arbiters.

Therefore, a more conservative model for an arbiter that more rigorously implements the QDI delay model, and captures the impact of arbitrarily different loads on the arbiter's outputs would include output buffers that have an unbounded delay. We refer to this as the **buffered arbiter** model. Specifically, the digital abstraction for the arbiter would artificially ensure that _u and _v are mutually exclusive low, and model u and v using ordinary inverters. The buffered arbiter model includes a transient state where u and v can both be high, consistent with Fig. 2.

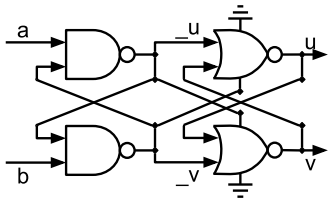

Fig. 3. Circuit diagram for the proposed Ideal Arbiter design.

Alternatively, an ideal arbiter that avoids the buffered arbiter model can be implemented by folding a latch into the instability filter as shown in Fig. 3. This forces the pull up networks of the arbiter's outputs to acknowledge the pull down network of the other, keeping them mutually exclusive. Importantly, the pass transistor logic implementing the instability filter has to remain intact. This is why _v is still the source node for the u↑ rule and _u for the v↑ rule. If more drive strength is necessary, the output signals may be further buffered while maintaining the ideal arbiter model by adding more latches. This arbiter will be expressed in circuit diagrams as a box labelled "iArb".

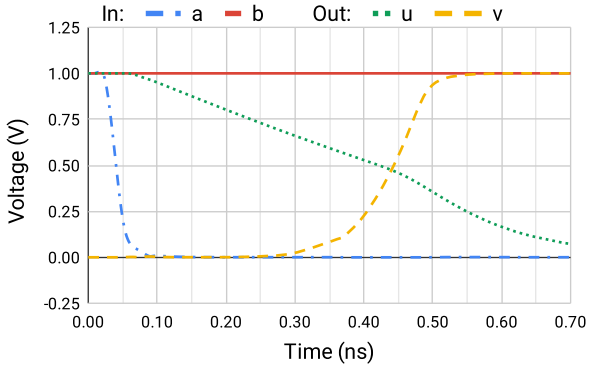

Fig. 4. SPICE waveform for the proposed Ideal Arbiter design.

Now in Fig. 4, the upgoing transition on v waits for the downgoing transition on u to pass the threshold voltage.

III. Existing Designs

The buffered and ideal arbiter models ultimately help to identify the timing assumptions beyond the strict QDI delay model that might be well-known by the authors of the work, but unclear to the readers. In subsequent sections, we will present designs that do not rely upon these timing assumptions for correct operation, thereby improving their reliability.

A. Bundling Merge

If we apply the buffered arbiter model to the bundling merge circuit from [1] in Fig. 5, it is possible for the circuit to go through an entire cycle, asserting and deasserting the grant request on S_r while uA or uB remains high. This is because w can be driven low following S_a↑ and therefore can skip the check for ¬uA ∧ ¬uB. As long as uA or uB continue to remain high, S_r and S_a will continue to complete full handshakes, causing A_a or B_a to toggle without A_r and B_r ever switching.

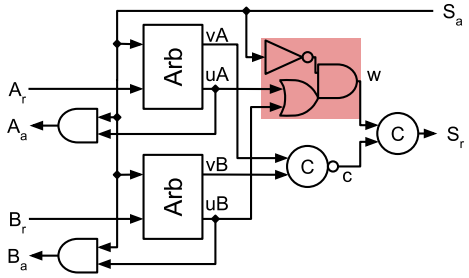

Fig. 5. Opportunistic Merge from [1].

However, the inverter that causes this failure cannot be removed. If it were, then the decomposition of w and c from the gate driving S_r would create a second failure mode. After A_r↑ causes S_r↑ followed by S_a↑ and the request on A_r is acknowledged, eventually causing uA↓, another request can appear on B_r, simultaneously causing uB↑. This could cause a glitch on w that can propagate out S_r. The inverter and AND gate from S_a into w successfully covers this glitch as long as the application of the atomic complex gate assumption to the driver of w is valid. This assumption requires the gate highlighted red in Fig. 5 to behave as if it were a single gate.

Ultimately, this means that the inverter in that gate must transition before $uA\downarrow$.

We remark that since [1] already assumes a fast local inverter through their application of the atomic complex gate assumption to the driver of w, the assumption of fast inverters in the implementation of the arbiter is likely reasonable in practice as long as an effort is made to implement that assumption in layout.

B. Greedy Arbiter

Applying the buffered arbiter model to the greedy arbiter from [2] in Fig. 6 breaks far more. This is because the environment is allowed to bypass various stages of the internal handshake. Suppose a request on A_r has already been given the grant and A_r is subsequently lowered, but S_r and S_e are still high. Then, as S_r is being lowered, a request on B_r can arrive causing a down-up glitch on S_r.

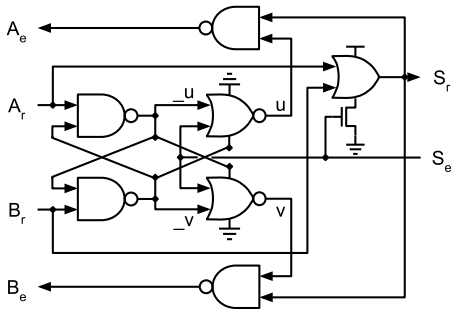

Fig. 6. Greedy Arbiter from [2].

If B_r happens a little later, and S_r subsequently transitions low, then the request on B_r will drive v high causing a down-up glitch to propagate out B_e.

Suppose B_r arrives even later and neither of these glitches happen. This means S_r remains low long enough for S_e to transition high. Then, as S_e transitions high, an up-down glitch could be generated on v since the request on B_r has simultaneously lowered $\_v$.

Finally, as documented in [3], the greedy arbiter found in [2] does not implement fairness and will revisit the same child repeatedly while deadlocking the other child.

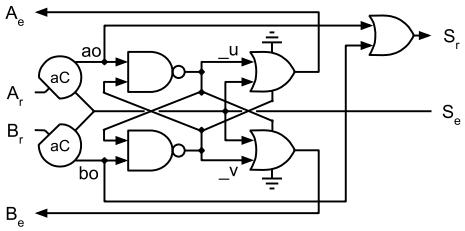

Fig. 7. Greedy Arbiter from [3].

The greedy arbiter from [3] in Fig. 7 already works under the buffered arbiter model, but still has other environment-dependent timing assumptions. Suppose a request arrives on A_r causing $S_r\uparrow$. Then, as S_e is lowered, another request arrives driving $B_r\uparrow$ and causing an up-down glitch on bo, then as long as $\_u$ has not transitioned low in response to the request on A_r, the glitch on bo could propagate through the arbiter and out B_e. Alternatively if a request never arrives on B_r, then when A_r is lowered, S_r will be lowered and S_e raised. Then, $S_e\uparrow$ can drive $A_e\uparrow$ without $\_u$ ever transitioning. So, a new request on A_r could cause an up-down glitch on $\_u$. If S_r goes up and S_e down in response to this new request, then that glitch will propagate out A_e. It should be noted that neither of these timing assumptions are hard to ensure in layout and that the authors of [2] and [3] made these assumptions intentionally to conserve transistors in the face of a severe restriction due to their application requirements.

IV. Maybe Execute Element

The behavior of the bundling merge can be succinctly described with a single selection statement. The selection statement is ultimately symmetric across the two input channels, meaning it behaves the same regardless of which channel request arrives first. Given two channels A and B using four phase communication protocols and one channel S using a two phase communication protocol, the bundling merge is as follows.

$$*[[\overline{A} \rightarrow S;A;S \mid \overline{B} \rightarrow S;B;S \mid \overline{A}\wedge\overline{B} \rightarrow S;(A\|B);S]]$$

However, the behavioral description of the greedy arbiter implemented in [3] is quite a bit more complex since it tries to remain symmetric. Instead of implementing a symmetric greedy arbiter, we implement an asymmetric one that looks similar to the bundling merge so that we can build both circuits with common building blocks.

$$*[[\overline{A} \rightarrow S;A;S \mid \overline{B} \rightarrow S;B;S \mid \overline{A}\wedge\overline{B} \rightarrow S;(A;B);S]]$$

Now, the grants are always served in the same order regardless of which request arrives first. This means that the only thing differentiating the bundling merge and the greedy arbiter is the behavior when A and B happen together. The bundling merge executes $A\|B$ while the greedy arbiter executes $A;B$. So lets just look at one of them.

$$*[[\overline{A} \rightarrow S;A;S \mid \overline{B} \rightarrow S;B;S \mid \overline{A}\wedge\overline{B} \rightarrow S;(A\|B);S]]$$

The first thing we can do is pull S out of the selection statement since it behaves the same regardless of the condition. The first action on S happens before any communication actions on A or B but after at least one of their associated probes. Therefore, we add an extra guard checking the probes of A or B and then move the first action on S so that it precedes the selection statement. The second action on S happens after all communication actions on A or B. Therefore, we pull the second action on S out to the other side of the selection statement.

$$*[[\overline{A}\vee\overline{B}];\ S;[\overline{A} \rightarrow A \mid \overline{B} \rightarrow B \mid \overline{A}\wedge\overline{B} \rightarrow A\|B];S]$$

Next, we can add in a redundant branch to the selection statement in preparation for later steps. The newly created branch $\neg\overline{A}\wedge\neg\overline{B} \rightarrow \texttt{skip}$ will never be executed due to the guard $\overline{A}\vee\overline{B}$ beforehand. This has a subtle side-effect of making the guards $\overline{A}\wedge\neg\overline{B}$ and $\neg\overline{A}\wedge\overline{B}$ unstable, requiring a synchronizer to implement the non-deterministic selection statement.

```
*[[Ā∨B̄]; S;
  [ ¬Ā∧¬B̄ → skip
  |  Ā∧¬B̄ → A
  | ¬Ā∧ B̄ → B
  |  Ā∧ B̄ → A‖B
  ];S]
```

This allows us to factor the selection statement into two, one for the channel A and one for B. This effectively pulls the parallel composition out of the selection statement. Note that the selection statements must remain non-deterministic because the probes $\neg\bar{A}$ and $\neg\bar{B}$ still may be unstable.

```
*[[Ā∨B̄]; S;
  ([¬Ā → skip | Ā → A] ‖
   [¬B̄ → skip | B̄ → B]
  );S]
```

Now, we can notice that $[\neg\bar{A} \rightarrow skip \mid \bar{A} \rightarrow A]$ is effectively a non-deterministic execute. If there is a request on A, then execute A. Otherwise skip. So, lets use process decomposition to factor this out. However, in order to correctly decompose these processes, we also have to keep the probes in $\bar{A} \vee \bar{B}$ in mind. To isolate A and B to the newly decomposed processes, we create two state variables uA and uB that keep track of the outcome of the non-deterministic selection statements. Then we can treat them as shared variables, checking their value to ensure the non-deterministic selection has resolved. To do this process decomposition, we introduce two new channels Sa and Sb that use four phase communication protocols. Finally, we can use the transformation described in [17] to implement the unstable guards on $\neg\bar{A}$ and $\neg\bar{B}$ using the stable guards on $\overline{Sa}$ and $\overline{Sb}$. This allows us to use an arbiter to implement the non-deterministic selection statements.

```
*[[S̄a → Sa | Ā → uA↑; [S̄a]; A; uA↓; Sa]] ‖
*[[S̄b → Sb | B̄ → uB↑; [S̄b]; B; uB↓; Sb]] ‖
*[[uA∨uB]; S;(Sa ‖ Sb);S]
```

Ultimately, the module we factored out $*[[\bar{S} \rightarrow S \mid \bar{A} \rightarrow uA\uparrow;\ [\bar{S}];\ A;\ uA\downarrow;\ S]]$ can be implemented very succinctly. If S_e arrives first, then we just do the complete handshake on S. If A_r arrives first, then we use a state variable uA to encode the output of the arbiter from the non-deterministic selection. Then, we can wait for S_e to give us the grant, and proceed to execute A by lowering A_e. Once A has completed its execution, as communicated by lowering A_r, we can reset the circuit, completing the handshakes on A and S in parallel.

```
*[[ Se → Sr↑;[¬Se];Sr↓
  | Ar → uA↑;[Se];Ae↓;[¬Ar];uA↓;(Ae↑‖Sr↑;[¬Se];Sr↓)
 ]]
```

This reshuffling leads to a very simple circuit using an ideal arbiter as shown in Fig. 8. One should note that because of the wire forks internal to the arbiter, any logic containing uA must also acknowledge S_r.

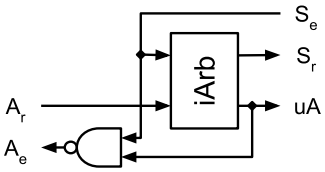

Fig. 8. Circuit diagram for the proposed Maybe Execute Element.

The circuit in Fig. 8 is called the Maybe Execute Element because when triggered on S it only executes the interfaced communication action on A if that communication is ready. If A is not ready, then the action is simply skipped and the trigger on S is acknowledged. Therefore, the action on A "may be executed".

To make this circuit easier to navigate, we introduce some syntactic sugar for CHP using a re-write rule.

```
*[[S̄a → Sa | Ā →uA↑; [S̄a]; A; uA↓; Sa]] ‖
*[… [uA] … Sa …]
```

Specifically, the above CHP is rewritten as follows:

```
*[… [Ā◦] … A◦ …]
```

This transforms the bundling merge specification to

```
*[[Ā◦∨B̄◦]; S;(A◦ ‖ B◦);S]
```

and the greedy arbiter specification to

```
*[[Ā◦∨B̄◦]; S;(A◦; B◦);S]
```

The parallel composition $A\circ \| B\circ$ as seen in the bundling merge specification, or the sequential composition $A\circ;\ B\circ$ as seen in the greedy arbiter specification is now just as simple in Fig. 9 as it would be using syntax directed translation [14].

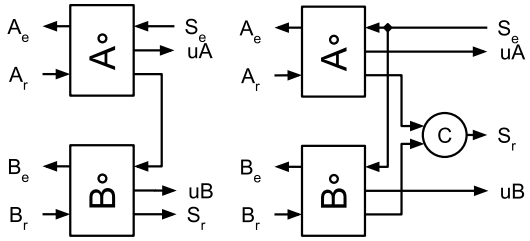

Fig. 9. Circuit diagram for the sequential (left) and parallel (right) compositions of the proposed Maybe Execute Element.

Its possible to implement the maybe execute circuit using buffered arbiters as in Fig. 10. The added transient state in the buffered arbiters desynchronizes uA and uB from Sr. Specifically, uA can go up before Sr goes down and Sr can go up before uA goes down. Now, the first case is not a problem because A_e is forced to wait for S_e anyway. However, the second case can cause an instability. Therefore, we have to use an asymmetric C-element to force S_r to wait for uA to transition to GND. This new C-element acknowledging $uA\downarrow$ can be easily folded into other logic in a larger design.

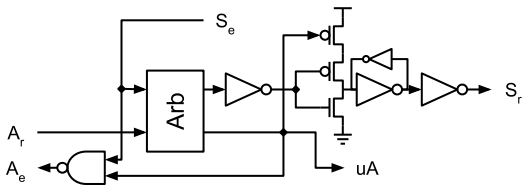

Fig. 10. Circuit diagram for the proposed Maybe Execute Element using the buffered arbiter.

Therefore, the only difference between the state transition diagrams of the ideal arbiter maybe execute and the buffered arbiter maybe execute is the first case that we identified. This just creates an extra state in Fig. 11 highlighted in red beyond the ideal arbiter's state transition diagram. Take note that this and further diagrams are rendered using the standard state transition diagram notation from the asynchronous literature, as found in [31], and is not a conventional finite automaton even though the graphical notation is similar.

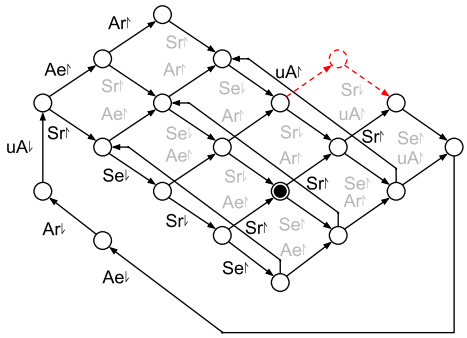

Fig. 11. State transition diagram of the proposed Maybe Execute Elements. The Buffered Arbiter approach adds the highlighted state.

It is possible to optimize this further by replacing the gate driving A_e with pass transistor logic. While it is more performant for two inputs, it does not scale well beyond that.

Lastly, note that for efficiency, this implementation assumes that the arbiter is fair as noted in [17]. If the arbiter is not fair, then it is possible for A to execute multiple handshakes before raising S_r. The opportunistic merge in [1] makes a similar assumption.

V. Greedy Arbiter and Bundling Merge

Circling back to the bundling merge and greedy arbiter, this transformation makes the specification for a bundling merge look very similar to a deterministic merge and a greedy arbiter look very similar to an alternating merge. The only departure from the deterministic specification is that we have to actively check to make sure at least one of the input channels is requesting a grant $[\overline{A}\circ\vee\overline{B}\circ]$. If not, then the circuit will simply busy wait instead of blocking, repeatedly and unnecessarily executing communication actions on S, wasting a lot of energy along the way.

$$*[[\overline{A}\circ\vee\overline{B}\circ];\ S;(A\circ\ \|\ B\circ);S]$$
$$*[[\overline{A}\circ\vee\overline{B}\circ];\ S;(A\circ;\ B\circ);S]$$

To see the busy-wait versions, look no further than back to the sequential and parallel composition circuits in Fig. 9. However, implementing the extra check for $[\overline{A}\circ\vee\overline{B}\circ]$ which makes it blocking requires adding only two extra transistors, as highlighted in Fig. 12, to what would otherwise just be an inverter. The transistor network that drives Sr in Fig. 12 and Fig. 14, is ultimately a generalized C-element with a weak staticizer. The cross-coupled inverters form a latch whose value is set by the pull-up and pull-down networks of the C-element. The weak backwards inverter is a staticizer, designed to keep the previous state of the C-element when both the pull-up and pull-down networks are off.

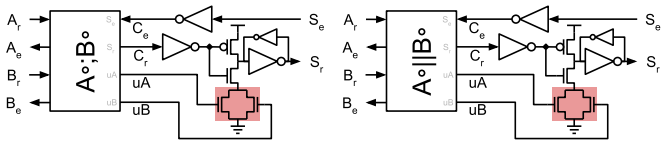

Fig. 12. Circuit diagrams for proposed greedy arbiter (left) and bundling merge (right).

The state transition diagram of this interface between C, S, uA and uB is shown in Fig. 13. Again, the extra states introduced in the buffered arbiter implementations are highlighted in red.

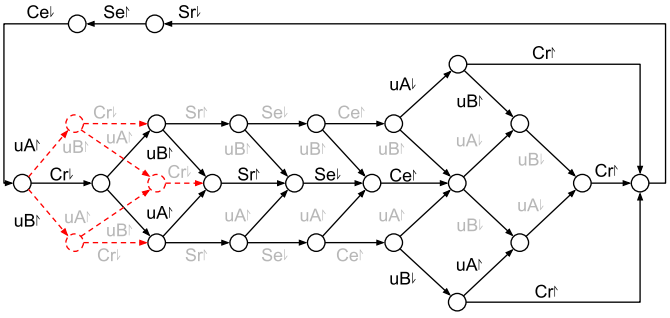

Fig. 13. State transition diagram of the proposed interface for the bundling merge and greedy arbiter. The Buffered Arbiter approaches add the highlighted states.

And with some peephole circuit optimization, we get the circuits presented in Fig. 14. Notice that no single gate in the greedy arbiter has a transistor stack length that is dependant upon the number of inputs. That means that one could scale it to an arbitrary number of inputs without ever introducing any gate trees. For the bundling merge, the generalized C-element driving Sr does grow with the number of inputs. However for the most part, that can be implemented as a tree of standard C-elements. The final C-element at the root of the tree would need to include the highlighted transistors attached to its ground node.

Furthermore, if the gates driving c, w, and Sr in [1] were merged together, the inverter on Sa would no longer be necessary. With the inverter removed, we would have successfully re-derived the bundling merge presented in this work. This similarity shows the power of our systematic approach to derive circuits of similar quality to carefully hand-crafted circuits in the literature.

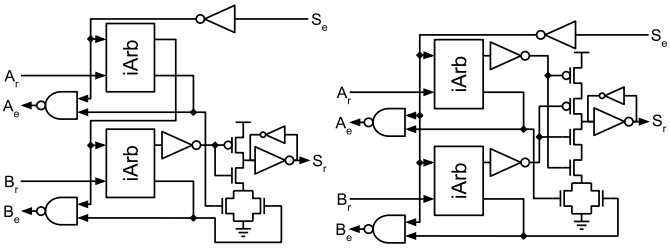

Fig. 14. Optimized circuit diagrams for the ideal-arbiter version of the proposed greedy arbiter (left) and bundling merge (right).

More generally, we can implement any arbiter of the form *[[wait expr];S;(compose expr);S] using the same basic template. For example, a one-writer two-reader lock where the writer gets priority is *[[$\overline{W}\circ v\overline{R_0}\circ v\overline{R_1}\circ$];S;($W\circ;(R_0\circ\|R_1\circ)$);S]. Alternatively, we could give the readers priority by modifying the composition expression: *[[$\overline{W}\circ v\overline{R_0}\circ v\overline{R_1}\circ$];S;(($R_0\circ\|R_1\circ$);$W\circ$);S]. Or we could require both readers to make a request before either of them get the grant by modifying the wait expression: *[[$\overline{W}\circ v\overline{R_0}\circ\wedge\overline{R_1}\circ$];S;($W\circ;(R_0\circ\|R_1\circ)$);S]. This last example demonstrates the power of this system to produce solutions to arbitration problems that cannot be otherwise constructed with the circuits found in the literature. In general, these types of solutions come from wait expressions that are not simply an OR across all input request signals.

When optimizing the buffered arbiter design in Fig. 15, take note that we do not have to wait for uA to go low before passing the grant off to the next request. This is because vA↑ requires that A_r is lowered thereby handing the grant back to us. We just need to make sure that uA and uB complete their transition before we complete the handshake. This allows us to merge the check for uA↓ and uB↓ into the C-element driving S_r making it a symmetric generalized C-element.

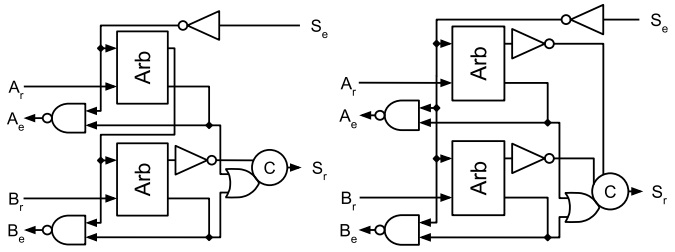

Fig. 15. Optimized circuit diagrams for the buffered-arbiter version of the proposed greedy arbiter (left) and bundling merge (right).

The process of extending this circuit to many inputs is not altogether obvious. There will still be a C-element tree driving Sr similar to the ideal arbiter version, with the special root node that checks the upgoing transitions of uA, uB, etc. However, the downgoing transitions of those signals must be checked at the leaves of the tree. This prevents the transistor stacks at the root node from growing too long.

Aside from the bundling merge circuit in [1], the circuits available in the literature rely upon hierarchical composition to scale to many inputs. This connects the child-grant channel A or B of one arbitration circuit to the parent-grant channel S of another. Furthermore according to the authors, scaling the flat composition in [1] above 3 inputs becomes dangerous due to their timing assumptions, and the standard cell library is unlikely to have the necessary gates for w in Fig. 5.

VI. Evaluation

We developed and evaluated all of these circuits using a set of in-house tools, a version of which is described in [22] and publicly available at [20]. We also verified correctness for all of the elements described in this paper with a brute force switch-level simulation which identifies instability, interference, and deadlock across all states. No environmental assumptions were required since this was verified using simple sources and sinks on the input and output channels. We automatically translated these specifications into SPICE netlists and verified their analog properties using Synopsys's combined simulator with VCS, a Verilog simulator, to simulate the testbench and HSIM, a fast SPICE simulator, to report power and performance metrics.

To evaluate the energy per operation and throughput of these circuits, we used a 1V 28nm process and protected each of the digitally driven channels with a FIFO of three WCHB [30] buffers isolated to a different power source to get more accurate results. All of the circuits are sized minimally with a pn-ratio of 2, and we use weak feedback staticizers for all of the C-elements in our designs. Circuitry necessary for reset was not included in any of the above descriptions. The performance values for the cited work is determined using the same method.

Fig. 11 shows the state transition diagram for the maybe execute element, covering all combinations of possible signal transitions under any possible timing. The figure was derived directly from the circuit, demonstrating that the maybe execute element has no switching hazards on any of the digital gates under any possible timing. Furthermore, the handshake on A always waits for the parent grant from S, and the parent grant is not returned until the handshake on A has completed. This guarantees that child grants remain mutually exclusive.

The circuit family we use guarantees that if two components are independently verified to be robust under all possible delay scenarios, and they only interact through delay insensitive handshake protocols, then their composition is also robust. Therefore, the compositions in Fig. 9 maintain timing robustness characteristics.

Finally, the state transition diagram in Fig. 13 was similarly derived, demonstrating that the bundling merge and greedy arbiter interface has no switching hazards on any of the digital gates under any possible timing. Specifically, all upgoing transitions on uA and uB must happen after C_e↓ and all downgoing transitions must happen after C_e↑, keeping the gate driving S_r stable.

Therefore, our designs suffer from none of the subtle timing issues we highlighted in Section 3. Meanwhile, our designs expose a more general strategy for composing arbitrary non-deterministic selection expressions and show good performance with reasonable energy requirements. Furthermore, our designs only require 2 gates beyond those typically found in commercial standard-cell libraries, the arbiter and the C-element executing the wait expression. These two cells can be easily implemented using standard techniques or automated layout [29].

In a real system, arbitration circuits are likely to be few and far between, so any enhancements demonstrated by our circuits are likely to have little to no effect on the system performance. Therefore, the goal of our analysis is simply to ensure that these circuits do not end up bottlenecking the system performance beyond what is already in the literature. Any performance enhancements demonstrated by our circuits are not generally a result of the systematic strategy we developed, but are instead a result of the peephole optimizations that come after. To be fair, it is surprising that circuits which more strictly implement the QDI delay model

can demonstrate any performance enhancement at all.

For the bundling merge `A‖B`, our buffered arbiter approach, with 68 transistors, operates 5% faster and consumes 4% less energy per operation than previous work [1] with 78 transistors. Our ideal arbiter version, with 69 transistors, is slightly worse than the normal arbiter design in every metric because it moves one of the acknowledgement checks earlier in the handshake.

Fig. 16 shows the performance and Fig. 17 shows the energy of each bundling merge design as they scale to many inputs. The high activity (top) plot assumes that all inputs are making requests and the grants are servicing those requests at max possible throughput while the low activity (bottom) plot assumes that only one input is making requests while the others are inactive.

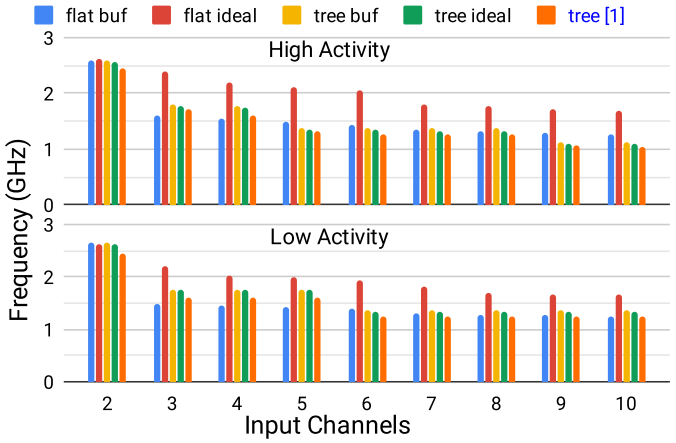

Fig. 16. High activity (top) and low activity (bottom) performance metrics for the bundling merge `A‖B` as it scales to many inputs.

When scaling the designs beyond 2 inputs with high activity, the flat ideal arbiter design is quickly superior in both frequency and energy. This is primarily because all of the nodes in the tree composition are unbuffered, so a request has to be passed all the way up and the grant back down the tree before the requesting process can be serviced. This incurs a significant delay that the flat compositions easily avoid.

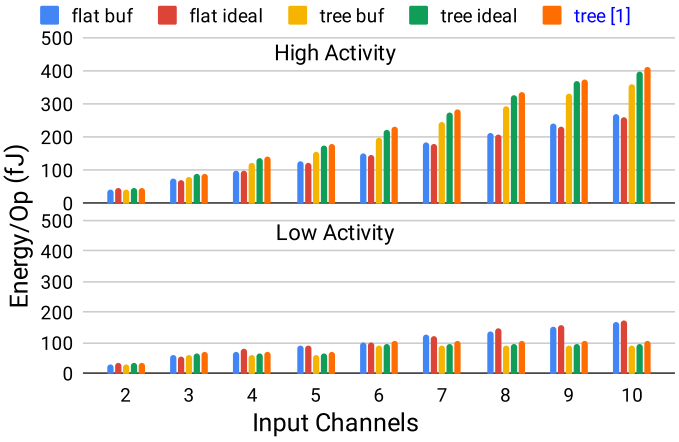

Fig. 17. High activity (top) and low activity (bottom) energy metrics for the bundling merge `A‖B` as it scales to many inputs.

At lower activity, the tree compositions will start to perform better because they have to check fewer nodes. More specifically, if `1` out of `N` nodes are making requests, the flat composition has to check all `N` while the tree compositions only have to check `O(log2(N))` nodes. In the low activity energy plots in Fig. 17, this is apparent because the tree designs use much less energy overall. However, the low activity frequency plots in Fig. 16 show that the flat ideal arbiter design remains superior to the tree designs. In a real system, if access to a shared resource is a bottleneck, and a bundling merge is used to arbitrate, then the performance gains from the flat ideal bundling merge can ultimately lead to an improvement in overall system performance.

For the greedy arbiter `A; B`, our buffered arbiter version, with 59 transistors, and our ideal arbiter version, with 65 transistors, both operate slower and consume more energy than [3] with 48 transistors and [2] with 35 transistors. This is to be expected since [3] and [2] both made significant timing assumptions, sacrificing reliability in order to fit their power and throughput budget.

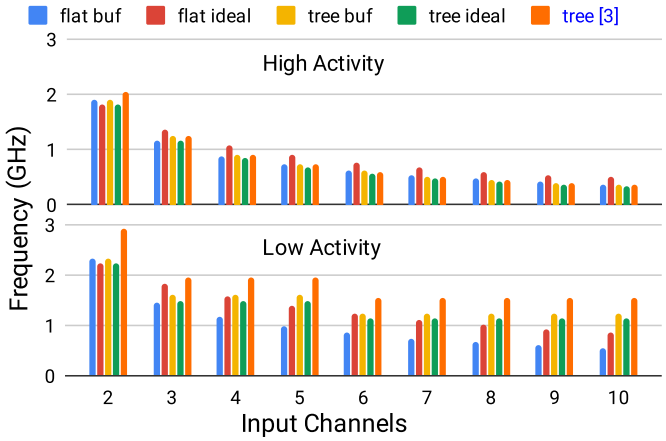

Fig. 18. High activity (top) and low activity (bottom) performance metrics for the greedy arbiter `A;B` as it scales to many inputs.

Fig. 18 shows the performance and Fig. 19 shows the energy of each greedy arbiter design as they scale to many inputs. Again, the high activity (top) plot assumes that all inputs are making requests and the grants are servicing those requests at max possible throughput while the low activity (bottom) plot assumes that only one input is making requests while the others are inactive.

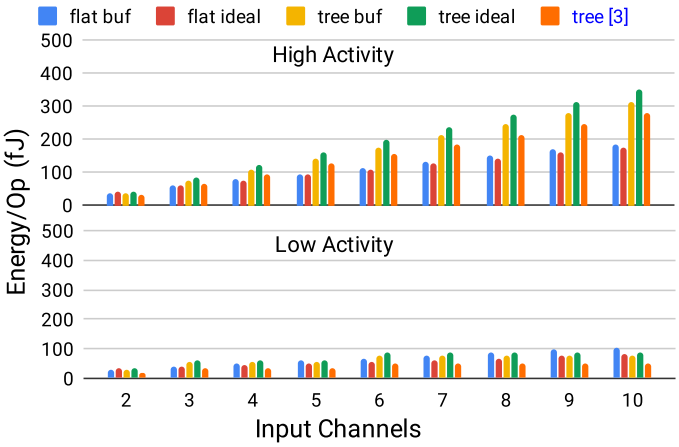

Fig. 19. High activity (top) and low activity (bottom) energy metrics for the greedy arbiter `A;B` as it scales to many inputs.

As the greedy arbiter is scaled beyond two inputs with high activity, the flat ideal arbiter design is still clearly superior to the tree designs in both frequency and energy.

However, the effect of lower activity is more significant. While the bundling merges check all `N` inputs in parallel, the greedy arbiter does so one at a time. This means that the tree composition will get a significant advantage to throughput beyond the flat compositions at lower activity levels. It is not possible to include [2] in this comparison because it deadlocks all but one input in this scenario.

VII. Conclusion

In this paper, we presented a robust implementation for a non-deterministic channel action and showed how it could be easily composed to generate robust implementations for bundling merges and greedy arbiters along with any other desired locking mechanism. Along the way, we elaborated on the difference between the buffered arbiter and ideal arbiter models and how those differences affect their respective circuit implementations. Finally, we evaluated these designs, showing better performing bundling merge and a similarly performing greedy arbiter.

Overall, the bundling arbiter and greedy arbiter are different problems. [2] and [3] simply present circuit diagrams for greedy arbiters rather than an approach that could be used to design a bundling merge as well. This is also the case for [1], where a bundling merge circuit is presented rather than an approach that could also be used to design a greedy arbiter. This paper is the first to present a unified approach to both problems.

Appendix

A. CHP Notation

Communicating Hardware Processes (CHP) is a hardware description language used to describe clockless circuits derived from C.A.R. Hoare's Communicating Sequential Processes (CSP) [15]. A full description of CHP and its semantics can be found in [27]. Below is an informal description of that notation listed top to bottom in descending precedence. For a complete discussion of the interaction between the handshake expansions of channel actions like send and receive and the composition operators, see [18].

Dataless vs Datafull Dataless expressions operate on node voltages while Datafull operate on delay insensitive encodings. Mixed expressions implicitly cast the datafull to dataless using the encoding's validity. Specifically, for a datafull expression `e` its positive sense `e` is cast to a validity check while its negative sense `¬e` is cast to a neutrality check. `null` is defined to be a neutral state of an encoding.

A **Channel** `X` consists of a **request** X_r and either an **acknowledge** X_a or **enable** X_e. The acknowledge and enable serve the same purpose, but have inverted sense. With these signals, a channel implements a network protocol to transmit data from one QDI process to another.

- **Skip** `skip` does nothing and continues to the next command.
- **Dataless Assignment** `n↑` sets the voltage of the **node** `n` to `Vdd` and `n↓` sets it to `GND`.
- **Assignment** `v := E` waits until the datafull expression, `E`, is valid, then assigns that value to the **variable**, `v`.
- **Send** `X!E` waits until the datafull expression `E` has a valid value, then sends that value across the **channel** `X`. Ultimately, a send is expanded into a handshake on its underlying signals. The standard four phase send on channel `X` is X_r `:= E;` $[X_a]$; X_r `:= null;` $[\neg X_a]$ for an acknowledge channel or X_r `:= E;` $[\neg X_e]$; X_r `:= null;` $[X_e]$ for an enable channel.
- **Receive** `X?v` waits until there is a valid value on the channel `X`, then assigns that value to the variable `v`. Ultimately, a receive is expanded into a handshake on its underlying signals. The standard four phase receive on channel `X` is `v :=` X_r; $X_a\uparrow$; $[\neg X_r]$; $X_a\downarrow$ for an acknowledge channel or `v :=` X_r; $X_e\downarrow$; $[\neg X_r]$; $X_e\uparrow$ for an enable channel.
- **Dataless Channel Action** If `X` is a dataless channel, then a send with an acknowledge channel is indistinguishable from a receive with an enable channel and a send with an enable channel is indistinguishable from a receive with an acknowledge channel. Therefore, we can simplify the syntax for the dataless send `X!` or receive `X?` to `X`.
- **Probe** $\overline{X}$`?` is used determine if the channel is ready for a receive action, returning the value waiting on the request X_r without executing the receive. $\overline{X}$`!` is used to determine if the channel is ready for a send action, expanding into either $\neg X_a$ given an acknowledge or X_e given an enable. For dataless channels, the syntax is simplified to $\overline{X}$.
- **Sequential Composition** `S; T` executes the programs `S` followed by `T`.
- **Parallel Composition** `S ‖ T` executes the programs `S` and `T` in any order.
- **Deterministic Selection** `[G1 → S1▯…▯`G_n `→` S_n`]` where G_i, called a guard, is a dataless expression and S_i is a program. The selection waits until one of the guards, G_i, evaluates to `Vdd`, then executes the corresponding program, S_i. The guards must be stable and mutually exclusive. The notation `[G]` is shorthand for `[G → skip]`.
- **Non-Deterministic Selection** $[G_1 \to S_1 | \ldots : G_n \to S_n]$ is the same as Deterministic Selection except that the guards do not have to be stable or mutually exclusive. If two or more evaluate to `Vdd` simultaneously, then one is picked arbitrarily (not necessarily random). In a circuit, this choice is implemented by a collection of arbiters and synchronizers. As discussed in Section 2, when two or more guards evaluate to `Vdd` simultaneously, it can cause a metastable state in the arbiter or synchronizer. This metastable state then resolves non-deterministically, giving the grant to one of the branches of the selection statement. Therefore, the digital model of this selection statement is also non-deterministic in such a condition.
- **Repetition** `*[G1 → S1▯…▯`G_n `→` S_n`]` is similar to the selection statements. However, the action is repeated until no guard evaluates to `Vdd`. `*[S]` is shorthand for `*[Vdd → S]`.

References

[1] Andrey Mokhov, Victor Khomenko, Danil Sokolov, and Alex Yakovlev. “Opportunistic merge element.” International Symposium on Asynchronous Circuits and Systems (ASYNC), Pages 116-123. IEEE, May 2015.

[2] Kwabena Boahen. “Point-to-point connectivity between neuromorphic chips using address events.” Transactions on Circuits and Systems II: Analog and Digital Signal Processing, Volume 47 Issue 5 Pages 416–434. IEEE, May 2000.

[3] Kwabena Boahen. “A burst-mode word-serial address-event link-i: transmitter design.” Transactions on Circuits and Systems I: Regular Papers, Volume 51 Issue 7 Pages 1269–1280. IEEE, July 2004.

[4] Mark Greenstreet. “Real-time merging.” Advanced Research in Asynchronous Circuits and Systems. 1999.

[5] Thomas Chaney, and Warren Littlefield. “The glitch phenomenon.” Computer Systems Laboratory, Washington University, Saint Louis, MO, Technical Memorandum 10, 1966.

[6] Thomas Chaney, and Charles Molnar. “Anomalous behavior of synchronizer and arbiter circuits.” Transactions on Computers, Volume 100 Issue 4 Pages 421-422. IEEE, 1973.

[7] Nabil Imam and Rajit Manohar. “Address-Event Communication Using Token-Ring Mutual Exclusion.” International Symposium on Asynchronous Circuits and Systems (ASYNC). IEEE, April 2011.

[8] Alain Martin. “Distributed mutual exclusion on a ring of processes.” Science of Computer Programming, Volume 5, Pages 265-276. 1985.

[9] Carver Mead, and Lynn Conway. “Introduction to VLSI Systems.” Addison-Wesley Longman Publishing Co., Boston, MA, 1979.

[10] Mika Nystrom, Rajit Manohar, and Alain Martin. “Method and apparatus for a failure-free synchronizer.” US Patent: US6690203B2, 2004.

[11] Fred Rosenberger, et al. “Q-modules: Internally clocked delay-insensitive modules.” Transactions on Computers, Volume 37 Issue 9 Pages 1005-1018. IEEE, 1988.

[12] Charles Seitz. “Ideas about arbiters.” Lambda First Quarter, Pages 10-14. 1980.

[13] Mishell Stucki, and Jerome Cox Jr. “Synchronization strategies.” Caltech Conference On Very Large Scale Integration, Pages 375-393. California Institute of Technology, Pasadena, CA, 1979.

[14] Steven Burns, and Alain Martin. “Syntax-directed translation of concurrent programs into self-timed circuits.” The Fifth MIT Conference on Advanced Research in VLSI, Pages 35-50. Cambridge, MA, 1988.

[15] Sir Charles Antony Richard Hoare. “Communicating Sequential Processes”. Communications of the ACM, pages 666-677. 1978.

[16] Sean Keller, Michael Katelman, and Alain Martin. “A Necessary and Sufficient Timing Assumption for Speed-Independent Circuits.” International Symposium on Asynchronous Circuits and Systems (ASYNC), pages 65–76. IEEE, 2009.

[17] Rajit Manohar, Mika Nystrom, and Alain Martin. “Precise exceptions in asynchronous processors.” Conference on Advanced Research in VLSI, Pages 16–28. 2001.

[18] Rajit Manohar. “An analysis of reshuffled handshaking expansions.” International Symposium on Asynchronous Circuits and Systems. ASYNC 2001. IEEE, 2001.

[19] Rajit Manohar, and Yoram Moses. “Analyzing isochronic forks with potential causality.” International Symposium on Asynchronous Circuits and Systems (ASYNC). IEEE, 2015.

[20] Rajit Manohar. “ACT Toolset.” https://github.com/asyncvlsi/act, 2019.

[21] Rajit Manohar, and Yoram Moses. “Asynchronous Signalling Processes.” International Symposium on Asynchronous Circuits and Systems (ASYNC). IEEE, 2019.

[22] Rajit Manohar. “An Open-Source Design Flow for Asynchronous Circuits.” Government Microcircuit Applications and Critical Technology Conference. March 2019.

[23] Alain Martin. “On Seitz' Arbiter.” Computer Science Department at California Institute of Technology: Caltech-CS-TR-86, 1986.

[24] Alain Martin, Steven Burns, Tak-Kwan Lee, Drazen Borkovic, and Pieter Hazewindus. “The First Aysnchronous Microprocessor: The Test Results.” Computer Science Department at California Institute of Technology, CaltechCSTR:1989.cs-tr-89-06, 1989.

[25] Alain Martin. “Programming in VLSI: From communicating processes to delay-insensitive circuits. No. CALTECH-CS-TR-89-1.” Computer Science Department at California Institute of Technology: CaltechCSTR:1989.cs-tr-89-01, 1989.

[26] Alain Martin. “The limitations to delay-insensitivity in asynchronous circuits.” Sixth MIT Conference on Advanced Research in VLSI, Pages 263–278. Cambridge, MA, 1990.

[27] Alain Martin. “Synthesis of Asynchronous VLSI Circuits”. Computer Science Department at California Institute of Technology: Caltech-CS-TR-93-28, 1991.

[28] Jon Tse, and Derek Lockhart. “An Asynchronous Constant-Time Counter for Empty Pipeline Detection”. jontse.com, 2009.

[29] Robert Karmazin, Carlos Tadeo Ortega Otero, and Rajit Manohar. “celltk: Automated layout for asynchronous circuits with nonstandard cells.” International Symposium on Asynchronous Circuits and Systems. IEEE, 2013.

[30] Andrew Matthew Lines. “Pipelined Asynchronous Circuits.” California Institute of Technology, 1998.

[31] Alexandre Yakovlev, Luciano Lavagno, and Alberto Sangiovanni-Vincentelli. “A unified signal transition graph model for asynchronous control circuit synthesis.” ICCAD. 1992.

Ned Bingham is a PhD student at Yale. He received his B.S. (2013) and M.S. (2017) from Cornell. During his Masters, he designed a set of tools for working with self-timed systems using a control-flow specification called Handshaking Expansions. Currently, he is researching self-timed systems as a method of leveraging average workload characteristics in general compute architectures. Between his studies, he has worked at Intel on Pre-Silicon Validation (2011, 2012), Qualcomm researching arithmetic architecture (2014), and Google researching self-timed systems (2016). (www.nedbingham.com)

Rajit Manohar is the John C. Malone Professor of Electrical Engineering and Professor of Computer Science at Yale. He received his B.S. (1994), M.S. (1995), and Ph.D. (1998) from Caltech. He has been on the Yale faculty since 2017, where his group conducts research on the design, analysis, and implementation of self-timed systems. He is the recipient of an NSF CAREER award, nine best paper awards, nine teaching awards, and was named to MIT technology review's top 35 young innovators under 35 for contributions to low power microprocessor design. His work includes the design and implementation of a number of self-timed VLSI chips including the first high-performance asynchronous microprocessor, the first microprocessor for sensor networks, the first asynchronous dataflow FPGA, the first radiation hardened SRAM-based FPGA, and the first deterministic large-scale neuromorphic architecture. Prior to Yale, he was Professor of Electrical and Computer Engineering and a Stephen H. Weiss Presidential Fellow at Cornell. He has served as the Associate Dean for Research and Graduate studies at Cornell Engineering, the Associate Dean for Academic Affairs at Cornell Tech, and the Associate Dean for Research at Cornell Tech. He founded Achronix Semiconductor to commercialize high-performance asynchronous FPGAs. (csl.yale.edu/~rajit)