A fast, asP*, RGD arbiter

Mark R. Greenstreet[*], Tarik Ono-Tesfaye[†]
{mrg,tesfaye}@cs.ubc.ca

Abstract

This paper presents the design of a high-throughput, low-latency, asP, RGD arbiter. Spice simulations for an implementation in a 0.8μ CMOS process show a request-to-grant delay of $0.74ns$ and a done-to-grant-delay of $0.4ns$. Maximum throughput of requests from a single client is one grant per $1.8ns$; if both clients make request aggressively, the arbiter can produce one grant per $1.2ns$. In addition to presenting a high-performance design, this paper examines trade-offs in performance driven design. In particular, logic delay seems to dominate metastability concerns when optimizing performance.*

1 Introduction

Ever since the pioneering work of Chaney and Molnar [4], designers of asynchronous circuits have been fascinated by the arbiter and its close cousin, the synchronizer. Much of this work has focused on the behaviour of these circuits under metastable conditions [12, 14, 9, 10, 16]. This paper addresses arbiter design from a performance-driven perspective. Because occurrences of prolonged metastability can be expected to be infrequent, our design optimizes performance under non-metastable operation: metastability must be taken into account to guarantee correctness, but performance is largely determined by the non-metastable case.

Metastability plays a much different role in synchronous and asynchronous design. For both arbiters and synchronizers, the time from an input event until a valid output is produced can be modeled as the sum of the time for signals to propagate through combinational logic of the circuit, t_c, plus the time for resolving any metastability that occurs, t_m. For the sake of comparing arbiters and synchronizers, we will assume a simple model where t_c is a constant and t_m is exponentially distributed with parameter τ. In this model, the probability that the circuit has not resolved an input event after t time units is $\min(e^{(t_c-t)/\tau}, 1)$. The value of τ depends on the dynamics of the circuit in a small neighbourhood near the metastable point, and t_c depends on the logical design of the arbiter or synchronizer.

In synchronous design, synchronizers are used to detect external events that are not synchronous to the system clock. A failure occurs if metastability is not resolved before the output of the synchronizer is sampled at a clock event. If a designer wants the probability of synchronization failure for a single synchronization event to be at most p_f, then the time allowed for synchronization, t_s, must be at least $t_c - \tau\log(p_f)$. Typically, p_f will be very small, and the minimum value for t_s is determined mainly by p_f and τ. Thus, synchronizer design focuses on circuit performance when the synchronizer is near its metastable point.

Self-timed circuits can continue computation as soon as the arbiter produces an output. In the design presented in this paper, τ is roughly $0.2ns$ and t_c is roughly $0.6ns$. For more than 90% of all input events, $t_c > t_m$. For designs such as this one, performance is determined largely by the delays of the combinational logic, and metastability is a secondary concern.

1.1 Top-level design

This paper presents an asP* RGD arbiter. An arbiter receives requests from two clients and issues grants to them such that at most one client is granted at a time. In an abstract model, a request or a grant is an "event." We will discuss below possible representations of these events in a CMOS circuit. Thus, a client asserts a request event and waits for the arbiter to issue a grant event. To indicate that the client no longer needs the priviledge, the client issues a done event. Although the two clients can issue requests concurrently,

[*]Currently on leave at the University of California, Berkeley. This work was supported in part by Canadian NSERC research grant OGP-0138501 a collaborative research grant from BC-ASI and SUN Microsystems, and United States Army Research Office grant ARO-MURI DAA404-96-1-0341.

[†]Supported by an NSERC PGS-B fellowship.

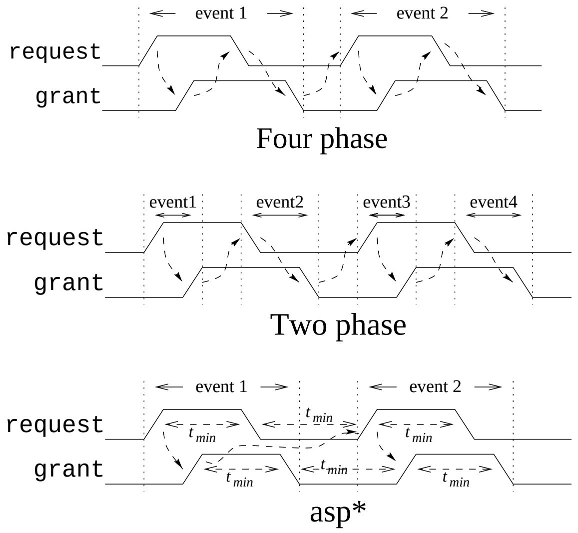

Figure 1. Self-Timed Handshakes

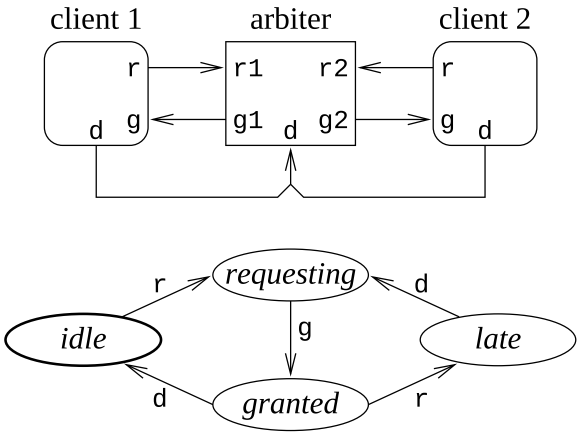

Figure 2. An RGD Arbiter

a client must be granted to issue a done. Thus, a minimal interface to an arbiter consists of two inputs for request events, two outputs for grant events, and one more input for done events. These observations lead to the RGD arbiter shown in figure 2.

Figure 2 shows the block diagram for an RGD arbiter [23] with two clients and the state transition diagram for a client. The sequence of operations for a client is to make a request with an event on `r`, wait for a grant event on `g`, and eventually release the privilege with an event on `d`. The arbiter implements a delay-insensitive interface [26]; in particular, when a client is in the *granted* state, it may issue the done for the current grant and the request for the next grant in either order. A granted client can make its next request before it asserts a done event; when this occurs, we say that the client is in the *late done* state. The possibility of a late done is a consequence of the delay-insensitive nature of the interface – exploiting the late done allows greater concurrency in the implementation. Mutual exclusion means that the arbiter issues only one grant event before receiving a done. This requirement can be formalized as:

$$\begin{array}{ll} \neg(& \mathit{Client1State} \in \{\mathit{granted},\ \mathit{late}\} \\ & \wedge \quad \mathit{Client2State} \in \{\mathit{granted},\ \mathit{late}\} \\ ) & \end{array} \tag{1}$$

We now consider the representation of events. Figure 1 depicts four-phase, two-phase, and asP* handshakes: logical constraints between transitions are indicated by unlabeled, dashed arrows; timing constraints are indicated by double arrows labeled with the a timing lower bound. In a four-phase handshake, and event is indicated by setting the corresponding signal to the high level. For each event, there is a corresponding reset operation, where the signal returns to the low level. For an arbiter with a four-phase interface, a client makes a request by raising its `request` signal. The arbiter grants the request by raising the corresponding `grant` signal. Typically, the done done event is indicated by lowering the `request` signal; this also accomplishes the reset operation for the `request`. Finally, the `grant` signal must be reset. According to the four-phase protocol, this reset must occur after the reset of the `request`. Thus, the four operations of the handshake must occur sequentially. This forces two round-trips of control transfer between the arbiter and the client for each grant issued. In other words, the control circuitry must operate at twice the arbitration rate; this rate doubling can limit performance.

With a two-phase protocol, only one `request` and one `grant` event occurs per arbitration, eliminating the overhead of the extra return-to-zero transitions of the four-phase handshake. However, the circuitry to implement a two-phase handshake can be complicated by the need to take both polarities of the signals into account. In particular, two-phase control circuits are often limited by the relatively slow speed of XOR gates.

With asP* signaling, events are represented by pulses [19], events are represented by pulses. This avoids the complexity of two-phase circuits because all pulses are of the same polarity. Unlike four-phase designs, there is no ordering constraint on the falling edge of pulses; instead, the circuit generating a pulse must ensure that the pulse satisfies a minimum width constraint, and the circuit receiving the pulse must acquire it within that time. We use asP* for its combination of concurrency and simplicity.

An asP* arbiter is a timed implementation of a delay-insensitive protocol. At the "events" level of abstraction, the interface to the asP* arbiter is delay-insensitive. However, each event is represented by a pulse, and the falling edge of the pulse is generated without any control from the receiver of the pulse. Thus, the sender is responsible for generating pulses of acceptable width, and these circits must be timed. In [19], asP* circuits are presented for datapaths. These circuits had both upper and lower bounds for pulse widths. This paper presents a high-performance asP* arbiter. Our design imposes lower bounds on the widths of request and done pulses (the inputs to the arbiter), but imposes no upper bounds. Our design guarantees both upper and lower bounds for the widths of grant pulses (the outputs from the arbiter) that are independent of the input pulse widths.

1.2 Performance

Our goal is to design a high-performance, asP* RGD arbiter. We quantify this performance with four measures:

Request to grant latency:
This is the time between when a client makes a request and when it receives a grant, assuming no contention from the other client. This provides an estimate of performance when both clients only make infrequent requests. Because there is no contention, metastability does not arise in this case.

Done to grant latency:
This is the time between when a client issues a done and when the next grant is issued, assuming that there is a pending request from at least one of the clients. This provides an estimate of performance when clients hold a grant for a long time making it likely that another request is waiting whenever a grant is released.

An RGD arbiter can be designed in a pipelined fashion, allowing contested requests to be resolved while waiting for a done. Thus, if clients hold the grant for a long time, metastability can be resolved with high probability before the done event. Thus, metastability does not have a strong impact on performance in this case.

Throughput of uncontested requests:
For this measurement, one client issues requests as quickly as it can and the other client issues no requests. This models a design where, for example, the arbiter is used to handle external interrupts for a self-timed processor. In this scenario, an arbitration is performed before each instruction is issued to check for an interrupt. Typically, interrupts are rare and most instructions receive a grant without contention from the other client. Thus, the performance for such a design is approximated by this uncontested request scenario. Because there is no contention, metastability does not arise in this case.

Throughput of heavily contested requests:
For this measurement, both clients issue requests as quickly as possible. This models a design where both clients are heavily active, such as the counterflow pipeline processor (i.e. CFPP) [23].

Under these conditions, metastability may occur before the first grant; then, successive grants alternate between clients. Thus, metastability has little significance in long-term throughput. In an application such as the CFPP, some instructions such as branches and stores do not produce results. Thus, more requests will be made by the instruction-pipeline than by the result pipeline. Thus, metastability may arise occasionally in such a design. As argued above, this should not have a significant impact on performance.

Our performance measures are based on SPICE simulation using model parameters provided by the Canadian Microelectronics Corporation for their 0.8μ service. Our simulations are based on schematic level design; thus, layout parasitics are not modeled. We normalize our measurements using a "typical gate delay." For this purpose, we consider the delay of an inverter driving four inverters of the same size to model a typical gate. SPICE simulations gave a value of $0.4ns$ for a typical gate delay.

1.3 Contributions

This paper presents the design of a high-performance, RGD arbiter. In particular, our design achieves a request-to-grant latency of 1.9 gate delays, a done-to-grant latency of 1 gate delay, an uncontested request throughput of one grant per 4.5 gate delays, and a contested request throughput of one grant per 3 gate delays.

The design of an asP* arbiter is inherently a timed design, and we make widespread use of concurrency to improve throughput. This leads to a design where it is difficult to find an intuitively convincing argument that the design is correct. Self-timed circuits have been popular examples for researchers in timed-automata [11, 24, 2]. We attempted to use the Mocha tool [1] to verify the correctness of our design; however, we have not been successful to date. For this approach

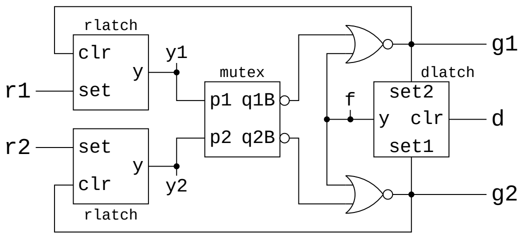

Figure 3. Block diagram of simple arbiter

to verification, considerable effort is required to find abstractions and write hand-crafted models that can be verified without running out of memory. Given the interest of the verification community in timed models, we believe that our design may be of interest to others in the verification community who wish to compare their methods with the ones described here.

2 Design

Figure 3 shows the implementation of a simple asP* RGD arbiter that provides the basis for our design. The `rlatch` is a positive edge-triggered R/S latch that converts pulse events on the request signals, `r1` and `r2`, to level signals. Each latch is set by a request event and cleared by the corresponding grant. The internal `mutex` provides mutual exclusion between the two clients. The `dlatch` converts done pulses to level signals, and the NOR gates ensure that a grant is issued only when the corresponding client has won the internal mutex and done events have been received for all previous grants.

Initially, the `r1`, `r2`, and `d` signals are low, and the `dlatch` and `rlatch`'es are reset: `y1`, `y2`, and `f` are low; `q1B` and `q2B` are high; thus, `g1` and `g2` are low. Consider the sequence of events that occurs when a request is received on `r1`. This sets the upper latch, bringing `y1` high. If no contesting request is incoming on `r2`, then `q1B` goes low after a short delay. Next, `g1` goes high, providing a grant for the request. Furthermore, the rising edge of `g1` sets the `dlatch` bringing `f` high, and then `g1` returns low. This is how the NOR gates and `dlatch` convert the level outputs of the internal mutex to pulse signals. Furthermore, the rising edge of `g1` also clears the `rlatch` returning `y1` to a low level and `q1B` to a high level.

At this point, client 1 is in the granted state. From here, client 1 can issue a done which will return the arbiter to the initial state, or either or both clients can issue new requests. In the latter case, the new request(s) can be resolved by the internal mutex while waiting for the done event. This pipelining allows the next grant to be issued quickly once a done event is

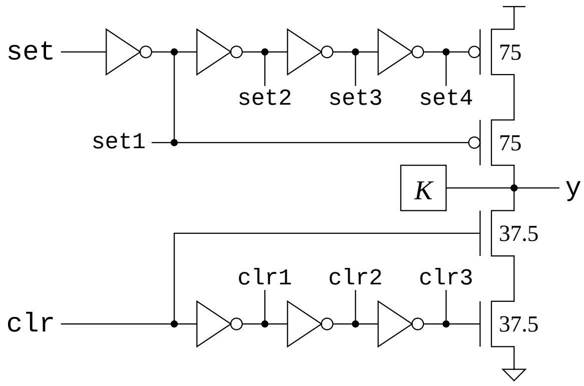

Figure 4. The `rlatch`

received.

Several optimizations to this design are described in the next subsection, and section 3 gives performance measurements from SPICE for the optimized design.

2.1 Details and optimizations

Appendix A contains a complete SPICE description of our design. Here, we describe the design and explain some of the trade-offs and rationale.

Figure 4 shows the `rlatch` design. Transistors are labeled with their shape factors. The box labeled K is a "keeper" circuit consisting of two inverters whose transistors each have shape factors of 2. Throughout our arbiter design, we use relatively large shape factors for most transistors. Typically, an arbiter will be used in the control path of a design and control a large number of circuits in the data-path. We believe that this justifies our use of large transistors; even larger devices may be used in designs with large data-paths.

When `clr` makes a low-to-high transition, `clr3` is initially high, and both n-channel transistors connected to the `y` output are conducting. This brings `y` low, thus clearing the latch. After the delay of the three lower inverters, `clr3` drops low providing the edge-triggered behaviour desired for the latch. In a similar manner, a low-to-high transition of `set` results in a high value on the `y` output. It is largely through the use of these edge-triggered latches that we obtained a design that does not have restrictions on the maximum input pulse widths.

Some care must be exercised with regard to the rise times of the `set` and `clear` signals. If these transitions are too slow, then the transition of `y` may be disabled before `y` has reached the desired logical level. In our simulations, the grant signals had rise times of roughly $0.55ns$ when loaded with a load equivalent to that of the request inputs. Based on this rise-time for

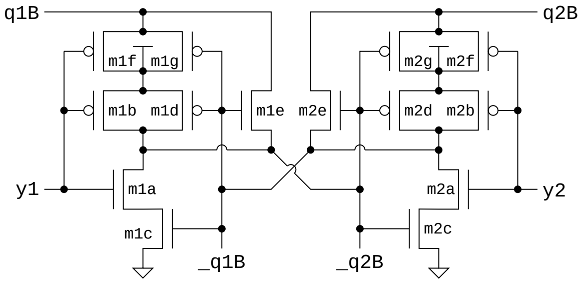

Figure 5. A two-NAND mutex

the grant, we used a rise time of $0.6ns$ for the request. We then verified that the latches functioned correctly when the load on the grant was increased by a factor of 5 or the rise time of the request inputs was doubled. We observed no adverse effects and concluded that these latches are adequately robust.

The starting point for the internal mutex of our design is the cross-coupled NAND gate design shown in figure 5. It is similar to the design in [?]. Transistors `m1a` through `m1d` form the NAND gate for client 1, and transistors `m2a` through `m2d` form the NAND gate for client 2. When the arbiter is in the idle state, both `y1` and `y2` are low, and therefore `_q1B`, `_q2B`, `q1B`, and `q2B` are high. If client 1 makes are request, `y1` will go high, which will bring `_q1B` and `q1B` low to indicate that client 1 can be granted. If both clients request concurrently, then the arbiter may enter a metastable condition where both `_q1B` and `_q2B` can remain arbitrarily long at voltages that do not correspond to well-defined logical values. The metastability filters prevent these ill-defined values from propagating to the rest of the circuit. The metastability filter for client 1 consists of transistors `m1e` through `m1g`. The key idea is that `m1e` will not conduct until the voltage at `_q1B` is lower than the voltage at `_q2B` by more than the n-channel threshold voltage. At this point, client 1 has won the arbitration, and `_q1B` will be low and `_q2B` will be high within a short amount of time. The transistors `m2e` through `m2g` form the metastability filter for client 2.

We made several changes to the design from figure 5 to achieve higher performance. Figure 6 shows half of the internal mutex. Two of these cells are cross-connected as shown in figure 7 to form the internal mutex. Note that `_qB` in figure 6 corresponds to `_q1B` (resp. `_q2B`) in figure 5; `qB` corresponds to `q1B` (resp. `q2B`); `q'` corresponds to `_q2B` (resp. `_q1B`). the signals `g`, `gB` and `g'` are connected to the grant for this client, the negation of the grant, and the grant for the other client as shown in figure `ref:arbiter`.

The first change optimization was the addition of a a "bypass" path to allow the rising edge of a request

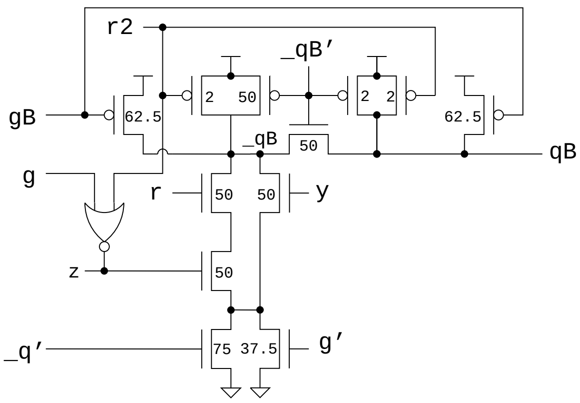

Figure 6. The `mutex/2`

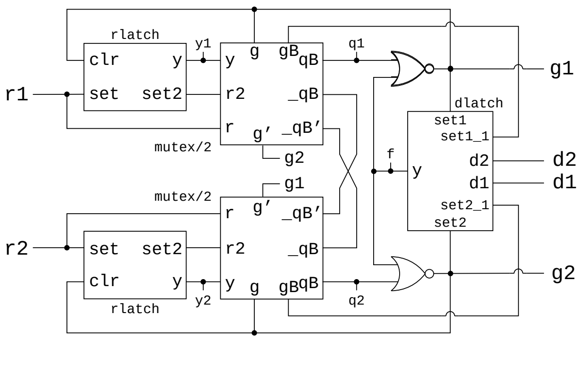

Figure 7. The optimized arbiter

to be applied directly to the arbiter without the incurring the delay of the `rlatch`. The n-channel transistors with gates connected to `r` and `z` provide this bypass. The `z` signal is high when a bypass is allowed. The `r2` signal is the request delayed by two inverter delays. Thus, the bypass is only enabled for rising edges of the request signal. This ensures that the design is robust in the presence of long request pulses. The `g` signal is the grant for the client driving this half of the mutex; `g` ensures that the internal mutex will be properly reset even if the next request arrives very quickly after the current grant. This design reduces the arbiter's latency for uncontested requests, but it complicates the dynamics of the arbiter and introduces several timing constraints. These issues are examined later in section 2.2.

The second change is to the pull-up network for `_qB`. In figure 5, there are two pull-up paths for `_q1B`: one of these is controlled by `y1`, the request; and the other by `_q2B`, the other side of the arbiter. We note that the pull-up controlled by `_q2B` is essential: it provides gain when the arbiter is in the metastable region. However, the pull-up controlled by `y1` can be replaced. First note that the iternal mutex is cleared when a grant is issued to the client. Thus, `g` causes the `y` and `z` signals to go low. Furthermore, `gB` controls the main pull-up for `_qB`. Finally, a small keeper transistor controlled by `r2` is included to mainitain `_qB` at a high level while waiting for a request event. The pull-up network for `_qB'` is similar; however, the transistor controlled by `_qB'` is only a small keeper. the pull-up network would become rather complicated and slow. Instead, we provide three, parallel pull-ups for the internal mutex. The p-channel transistor with its gate connected to `_qB'` provides the cross-coupling needed to resolve contested grants. The transistor with its gate connected to `gB` resets the mutex after a grant is issued. When the grant signal is high, `y` and `z` are driven low; so, there is no fighting in the `mutex/2` circuit. Finally, the p-channel transistor with its gate connected to `r` ensures static operation while waiting for a request.

The third change, is the addition of a n-channel device with its gate connected to the grant output for the other channel, `g'`, in parallel with the traditional pull-down connected to `_qB'`. The motivation for this is as follows. If the arbiter is receiving a large number of requests from both clients, then whenever client 1 is granted, the client 2 will be waiting for the next grant. Immediately after client 1 is granted, `y1` and the `gB` and `z` signals in the upper mutex are all driven low. This causes `_qB` in the upper mutex to go high. Because a request is pending from client 2, `y2` is high, and `_qB` in the lower mutex will now fall. In other words, when

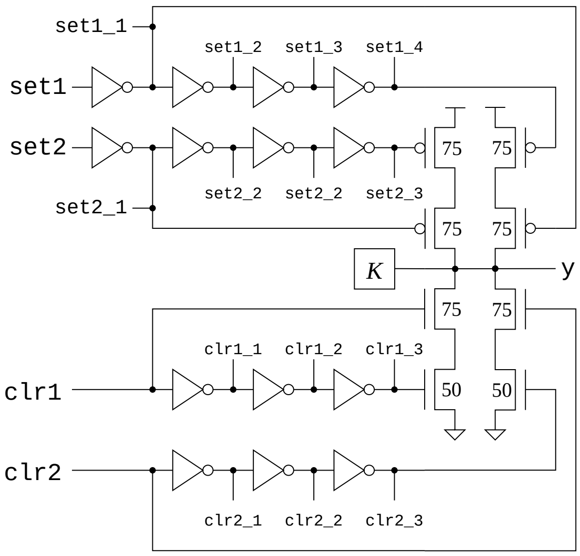

Figure 8. The `dlatch`

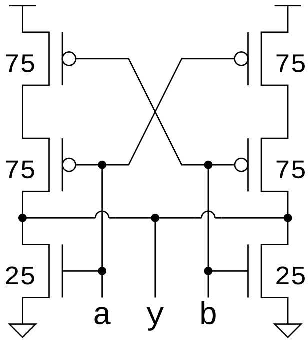

Figure 9. The output `NOR` gate

the arbiter is highly contested, it will issue grants to the two clients in alternation. Thus, when client 1 is granted, if there is a request waiting from client 2, we know that client 2 will win the next round of arbitration. The pull-down in the lower `mutex/2` controlled by `g1` allows this round to start earlier. In simulation, this increased the throughput under heavy contention by 15%.

Figure 8 shows the `dlatch`; its design is similar to that of the `rlatch`. The `dlatch` can be reset by a rising edge of either grant signal. Initially, we attempted to combine the two grant signals with a NOR gate. However, it is possible to issue a grant one gate delay after the other grant is dropped, and such operation causes runt pulses on the output of the NOR gate. Our solution was to employ separate pull-up networks and inverter chains for the two grants.

Figure 9 shows the output NOR gate. This NOR

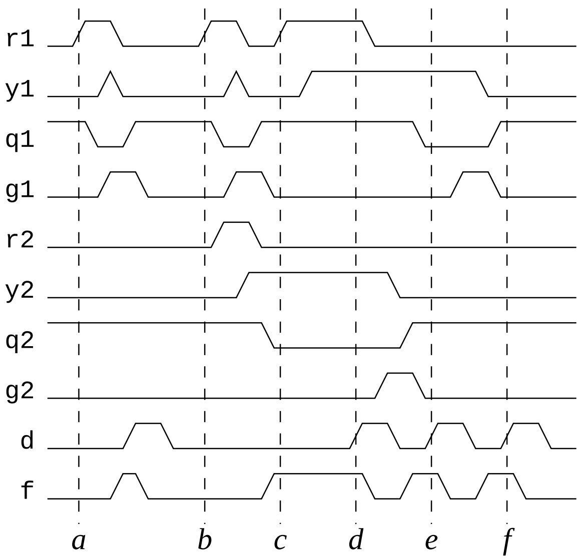

Figure 10. Example operation of the arbiter

gate is designed to minimize rise time of the grant signal: the delay from `q1B` (resp. `q2B`) falling to `g1` (resp. `g2`) rising is critical for all of our performance measures. Thus, this NOR gate has very large pull-up transistors. This has two other consequences. First, when one of the inputs of the NOR gate goes low, a significant amount of charge can be transferred from the node between a series pair of p-channel transistors and the output node, `y`. To avoid generating a spurious pulse on `y`, we split the pull-up into two parallel paths. This results in only half of the worst-case charge transfer when compared with a design with a single series pair of transistors. Second, this gives the NOR gates a relatively high switching threshold. This high threshold contributes to the low latency of the design.

It should be noted that when a grant signal goes high, there is a race condition for returning the grant to a low level. One path sets `f` high which causes the grant to go low. The other path sets the `qB` signal for that grant high which then causes the grant to go low. No matter which path wins the race, the grant will go low as desired. However, if the outcome is too imbalanced, then either `f` or `qB` may fail to go completely high. In our design, we found it was necessary to enlarge the pull-up transistor for `_qB` controlled by `gB` in the `mutex/2` circuit to ensure correct operation.

Figure 10 shows the arbiter handling three requests from client 1 and one request from client 2. At time a, client 1 issues its first request. The rising edge of `r1` causes `q1B` to fall and then `g1` to rise and grant the request. In figure 10, the client quickly issues a done, and the arbiter returns to its initial state. At time b, client 1 issues its second request, and client 2 issues a request immediately after. In this scenario, client 1 wins the internal mutex (i.e. `q1B` falls), and a second grant is issued to client 1. This grant causes client 1 to release the internal mutex, which is then acquired by client 2. However, this time client 1 continues to hold the privilege, and, at time c, client 1 makes another request, thus moving to the late done state. This request waits as a pending request for the internal mutex, as indicated by the high level on `y1`. When client 1 finally issues a done at time d, the arbiter can quickly issue a grant to client 2 because the internal arbitration has already been resolved. This pipelining in the arbiter contributes to the throughput of this design. At time e, client 2 issues a done, and client 1 receives its third grant. At time f, client 1 issues its third done, and the arbiter returns to its initial state.

2.2 Dynamics

By designing this arbiter for low-latency and high throughput, it exhibits dynamics that are unusual when compared with a design that tries to minimize the time-constant associated with the metastable state.

The internal mutex and the output NOR gates are designed to minimize the delays incurred when generating a grant event. Thus, the n-channel transistors in the pull-down network of the `mutex/2` and the p-channel transistors in the pull-up network of the output NOR gates are relatively large. This gives the internal mutex a low switching threshold and the output NOR gates high thresholds, which is just what is needed for low latency. However, this design prevents using the threshold of the NOR gate as a metastability filter. Instead, we used the more robust, differential filter, and incurred an extra delay propagating falling events of the `_qB` signals to `q1B` and `q2B`. Spice simulations showed that the more aggressive designs for the internal mutex and output NOR gates more than made up for this extra delay.

Several of our optimizations slow down the internal mutex when operating near the metastable point. First, the relatively small pull-up transistors in the mutex mean that the metastable point is at a lower voltage than the optimal one for maximizing the gain-bandwidth product of the circuit.

Second, the bypass path in the pull-down network increases the capacitances in the circuit, and this path will not remain active during prolonged metastability. Third, the keeper pull-up transistors controlled by `r2` will be conducting during metastability, shunting current out of the `_qB` nodes and lowering the gain or the circuit. Finally, we used relatively large, n-channel

client 2	client 1 bypass only	y only	both
bypass only	(1.82, 1.82)	(1.80, 2.04)	(1.82,1.80)
y only	(2.04, 1.80)	(2.02, 2.02)	(2.04, 1.78)
both	(1.80, 1.82)	(1.78, 2.04)	(1.80, 1.80)

Each entry gives the voltages on (_q1B, _q2B) at the metastable point with the bypass and y paths active as indicated.

Table 1. Effect of bypassing on the metastable point

transistors in the metastability filter. Smaller transistors would decrease the capacitive load on the _qB, and improve performance near the metastable point.

Although the bypass path involves three transistors in series, if the mutex is in a metastable condition, then the active pull-down path consists of two transistors: one controlled by y and the other controlled by _qB'. Under metastable conditions, the effective circuit topology (i.e. ignoring series transistors where at least on is in cut-off) is nearly identical to that of the simple mutex 5. The only difference is the slight shunting of current by the keeper transistors controlled by r2.

We note that the time constant of the metastable point is roughly 2π divided by the gain-bandwidth product of the circuit operating at the operating point. We used the unity-gain frequency as an estimate of the gain-bandwidth product. Spice simulations show that the unity gain point of our circuit is roughly 850MHz. If our optimizations are removed, the unity gain point increases to 1600 MHz. Thus, our design would be a poor choice for a synchronizer as it will take nearly twice as long to achieve the same reliability as a synchronizer designed to minimize the time-constant of the metastable point.

Another consequence of the bypasses is that the arbiter does not have a unique metastable point. Either mutex/2 can have either or both pull-down paths active. As shown in table 1, we observe various unstable equilibria at voltages between 1.78 volts and 2.04 volts depending on the state of the bypasses. The asymmetric saddle points violate the assumptions behind the metastability filter. However, we were not able to find any inputs that could evoke failures of the metastability filter. We hope that a more convincing verification of the circuit dynamics could be carried out along the lines described in [17]. Although this currently does not seem feasible, we are working on developing tools to perform such verification [6, 7].

3 Performance

Our performance measures are based on SPICE simulation using model parameters provided by the Canadian Microelectronics Corporation for their 0.8μ service. Our simulations are based on schematic level design; thus, layout parasitics are not modeled. We normalize our measurements using a "typical gate delay." For this purpose, we consider the delay of an inverter driving four inverters of the same size to model a typical gate. SPICE simulations gave a value of $0.4ns$ for a typical gate delay.

The CMC 0.8μ service is a 5 volt, n-well process. Our delay measurements are the time from the input signal crossing the 50% point (i.e. 2.5 volts) until the output crosses the 50% point. Our rise and fall time measurements are 5/3 times the time between crossing the 20% and 80% points (i.e. 1 and 4 volts). Our design tolerates very large pulse-widths; however, we assume that signal values may be held by dynamic storage during the high-period of a pulse. Keepers are used if a node must hold a signal for an arbitrarily long time. For the performance measurements reported here, we used request and grant pulses that were $1.2ns$ wide measured from the 50% point of the rising edge to the 50% point of the subsequent falling edge. The request and grant pulses had rise and fall times of $0.6ns$ making them slightly slower than the $0.5ns$ – $0.56ns$ rise and fall times that we observed for the grant signal. We load the grant with an inverter whose total gate area of $64\mu^2$ corresponds to the load at the request input of the arbiter (the done presents a load with a gate area of $60\mu^2$).

3.1 Request to grant latency

We measured the request-to-grant latency by starting with an arbiter in the idle state and applying a pulse to the r1 input. The time from the 50% point of the rising edge of r1 to the 50% point of the rising edge of g1 was $0.74ns$. This corresponds to 1.85 gate delays.

The critical path that determines request-to-grant latency is: r1 rises at $0ns$; _qB (in the upper mutex/2) falls, at $0.29ns$; q1B falls at $0.45ns$, and g1 rises at $0.74ns$. The longest delays are associated with the falling of _qB and the rising of g1. To minimize the delay along this path, we enlarged the pull-downs for _qB and q1B until increasing their sizes further caused a net slow-down due to the increased load. It is the large size of these transistors that gives the internal mutex a relatively low metastable point as discussed in section 2.2. Likewise, we enlarged the p-channel devices in the output NOR gate to minimize the `q1B` to `g1` delay. Here, the limitation was the "bounce" on `g1` when one of the inputs to the NOR gate dropped before the other. As mentioned described in section 2.1, we split the pull-up network for each NOR gate into two paths to minimize this bounce. With this design, our simulations revealed a worst-case bounce of $0.17V$ for about $0.6ns$. When the arbiter is used in a specific design, the clients may be tolerant of large variations in the grant signal. In this case, even stronger pull-ups could be used to decrease the request-to-grant delay.

3.2 Done to grant latency

We measured the done-to-grant latency by starting with an arbiter in the idle state, applying a pulse at the `r1 input`, waiting for the `g1` pulse, applying another pulse at the `r2` input, waiting $5ns$ for the circuit to settle, and then applying a done pulse. The time from the 50% point of the rising edge of `d1` to the 50% point of the rising edge of `g2` was $0.40ns$. The critical path that determines done-to-grant latency is: `d1` rises at $0ns$; `f` falls, at $0.13ns$; and `g2` rises at $0.40ns$. Not surprisingly, most of the delay is incurred in the NOR gate, and the comments made about request-to-grant latency apply here as well.

3.3 Throughput of uncontested requests

For uncontested requests, the circuit can issue a grant every $1.8ns$. We measured this by leaving the `r2` input low and applying pulses of different frequency to the `r1` input. Done signals are issued $1.6ns$ after the requests. Performance was limited by the rate at which the grant signal could transition. In this simulation, the next request is issued $0.56ns$ after the current grant, and the grant signal swings between $0.5V$ and $4.5V$. If the request is issued earlier, then grants are issued more often, but the grant signal makes smaller and smaller excursions. Thus, we decided that the minimum grant to next request delay should be $0.56ns$ (i.e. 1.4 gate delays) which seemed reasonable as it is unlikely that a client will have fewer than two gates in the path from grant to request.

The critical path that determines the maximum throughput is: `g1` rises at $0ns$, `r1` rises at $0.56ns$, the `set1` signal of the upper `rlatch` falls at $0.66ns$, and `y1` rises at $0.91ns$. For maximum throughput, the bypass path in the `mutex/2` is not employed because the `z` signal is still low when the next request is asserted. The path to the next grant is completed with `_q1B` dropping at $1.28ns$ `q1B` dropping at $1.54ns$. Finally, the next grant is issued at $1.8ns$.

3.4 Throughput of heavily contested requests

In the case where both clients are issuing requests, the circuit can issue a grant every $1.2ns$. For this measurement, we applied a pulse on the `r1` input every $2.4ns$ and issued a done on `d1` $2.0ns$ after each request. On `r2` we issued a request also every $2.4ns$, $1.2ns$ after the `r1` pulses. Similarly, `d2` pulses were applied every $2.4ns$, $2.0ns$ after the corresponding `r2` event. The critical path is: `g1` rises at $0ns$, `g1B` drops at $0.23ns$ and `_q1B` rises at $0.61ns$. which enables `q2B` to drop at $0.93ns$ in response to an earlier request on `r2`. Then, `g2` rises at $1.2ns$ Again, the `done` signals are not in the critical path. The path from the rising edge of `g2` to the next rising edge of `g1` is symmetrical.

4 Related Work

Although arbiters and asynchronous design are linked in the minds of many, much work in asynchronous design focuses on arbiter-free design. For example, the Caltech Asynchronous Microprocessor [15], the self-timed rings of Williams [27] and Sparsø and Staunstrup [22], and the Amulet2e [5] are all arbiter-free designs. Likewise, much of the performance analysis that has been published focuses on designs without arbiters: for example, the analysis of self-timed pipelines in [8] and [28], or that of meshes in [25] and [20]. Burns' analysis technique for the Caltech processor [3] considered probabilistic instruction mixes but was still restricted to an arbiter-free design style.

The recent work of Xie and Beerel [29, 30] based on Markov-chains provides a framework in which the impact of arbiter design on system-level performance could be studied. This approach could offer a more quantitative assessment of the relative significance of combinational logic delay and metastability delay than was presented in section 1 here. In particular, their discrete probability models correspond nicely to an enumeration of various paths through combinational logic, and the exponentially distributed random variables in their model provide a natural model for delays associated with metastability.

Kinniment *et. al.* have described a novel A-D design based on the observation that metastability should occur at most once in the conversion of any sample [13]. Given a time-budget for converting a sample, the delay for combinational logic in the design is incurred for each bit, and a safety margin for metastability is only added once. This design may provide an intermediate point between the combinational-logic-dominant nature of arbiter design and the metastability-delay-dominant nature of synchronizers. A similar trade-off occurs in Seizovic's pipeline synchronizer [21].

5 Trade-offs, Limits, and Opportunities

This paper has presented the design of a fast, asP*, RGD arbiter. Our design is certainly not the last word in fast arbiter design. In this section, we examine a few of the trade-offs that influenced our design, describe some limitations that seem to be inherent in any arbiter design, and mention some opportunities for faster implementations.

5.1 Trade-offs

A key trade-off in our design is one of throughput versus latency. We designed our arbiter for use in a context where heavy demands are made on it, and thus gave higher priority to throughput considerations. Probably the most important consequence of this trade-off is to separate the metastability filter from the output NOR gates that provide the falling edges of the grant pulses. Had we integrated the pulse generation logic into the metastability filter, we could have reduced the request-to-grant latency at a cost of degrading maximum throughput. By separating these into two separate operations, the internal mutex can resolve the next round of arbitration before the arbiter has received a done for the current round: operations are pipelined between the internal arbiter and the external interface. As noted in sections 3.3 and 3.4, this pipelining contributed to the throughput performance of our arbiter.

Another choice that favored throughput over latency is the use of the `g'` n-channel device in the `mutex/2` circuit. As explained in section 2.1 this feature significantly increases the throughput during heavy contention. However, this choice also increases the request-to-grant latency by increasing capacitances in the `mutex/2` and on the `g1` and `g2` signals.

The "bypass" feature in the `mutex/2` dramatically decreases request-to-grant latency. Without the bypass, the request-to-grant latency is $1.2ns$, as each request must pass through four levels of logic before a grant is issued: two in the `rlatch`, one in the `mutex/2`, and one in the output NOR gate. Adding the bypass reduces this latency by 38% at an expense of slightly lower maximum throughputs. We felt that this trade-off was justified to avoid embarrassing latency numbers.

A higher-level trade-off is the choice of a handshaking convention. The advantage of asP* is that all events are of the same polarity and that falling edges of pulses do not require communication events. Both of these features simplified the implementation and improved performance. However, this requires the output NOR gates that provide the falling edge of the grant signal. As noted above, these gates introduce a pipelining into the design that increases throughput at a cost of greater latency. In this way, asP* favors throughput over latency.

Another high-level choice was not to assume a maximum pulse width for request and grant pulses. The inverter chains in the `rlatch` and `dlatch` circuits ensure that these latches are edge triggered. If the environment guaranteed short request and done pulses, these circuits could be simplified with a resulting improvement in performance.

5.2 Limitations

Stepping back from the details of our design, we can measure its performance in terms of the number of logic levels (i.e. channel connect regions) that an event must pass through for each of our performance measures. The request-to-grant path has two logic levels as does the done-to-grant path. Six levels of logic must be traversed between successive grants to the same client. The path from issuing a grant to one client and issuing it to the next has four levels of logic. These values seem minimal given the conventions of asP* and the observation that it is impractical to reliably propagate a pulse that is narrower than three gate-delays.

The request-to-grant path will naturally have two levels: one for the internal mutex and one for some kind of metastability filter. Furthermore, if request and grant pulses are of the same polarity, then this path must have an even number of levels – again, two is minimal. Polarity arguments also indicate that the minimal done-to-grant path is two levels. To reliably propagate a pulse, its high and low portions must each be at least three gate-delays wide. Thus, the six levels per grant pulse to the same channel is minimal.

We assume that the arbiter design is symmetric with respect to the two clients. Polarity arguments indicate that the minimal grant to other grant path must have an even number of logic levels. If this path is traversed twice, it must have at least six levels as noted above. Thus, with the added assumption of symmetry, the four level path from grant to other grant is minimal.

These observations suggest that our design is minimal. Improvements are certainly possible by adjusting transistor sizes, choosing different trade-offs, or changing details of the logic design. While such changes might result in improvements of up to 20-30%, larger improvements will require a change in the basic assumptions.

5.3 Opportunities

Based on the observations above, we can suggest possible paths for design where we do not know how to design such an arbiter, but where such a design might offer improved performance.

One possibility is to move more responsibility to the clients. For example, if the metastability filter is integrated into logic of the client, an arbitration could be performed in only one logic level. Likewise, if additional timing constraints are known about the client (e.g. maximum pulse width, minimum and/or grant-to-done time, etc.) it may be possible to design a faster arbiter whose correctness depends on these properties of the client.

The design of an arbiter that operates directly on complementary signals offers a possibility of circumventing our polarity arguments and exploiting transition signaling as a faster alternative to asP*. The "propeller" and "four-mutex" designs presented in [18] show some promise for this approach. However, in these two designs, each arbitration is resolved by a single, cross-coupled stage with the stage being chosen by the parity of the done input and levels of the request inputs. This selection of an internal mutex is on the critical delay path for both designs. Furthermore, the internal mutex has a single rail output, while the signaling convention requires a complementary pair for the grant signal. This introduces an extra inversion and thus an extra level of logic for one of the grant lines. To obtain a minimal design, we would need a pair of complementary internal mutex elements that are coupled in such a way as to ensure that they resolve metastability consistently. Or, perhaps, a completely different approach is possible. The design of a complementary mutex seems to raise new questions about the dynamics of arbiters and new challenges for circuit design.

6 Summary

This paper has presented a design for a high-throughput, low-latency, asP*, RGD arbiter. We employed asP* signaling and the RGD convention because they offer greater concurrency than is possible with a traditional level arbiter. In particular, we have shown how the RGD design allows a pipelining between the internal mutex and the arbiter's interface circuitry. This allows the next round of arbitration to be resolved inside the arbiter while the grant and done events for the current round are being exchanged with clients. We exploit this pipelining to increase the throughput of our design.

More generally, we have described quantitative criteria for evaluating arbiter designs based on latency and throughput measures. We have argued that although metastable behaviour must be taken into account to produce a correct design, metastability has a relatively small impact on performance. The delays of an arbiter can be divided between those for combinational logic and those due to any metastability that may occur. For most applications, the logic delay will dominate the metastability delay and performance-oriented design should tend to emphasize logic over metastability. For example, our mutex design includes many features to reduce logic delay while doubling the time constant associated with resolving metastability compared with a traditional (cross-coupled NAND gates) design. Our simulations indicate that these choices contributed significantly to the performance of our design.

Our design is highly concurrent and depends critically on the dynamics of its mutex for correct operation. We have performed extensive simulations and believe that our design is correct. We also believe that more formal approaches to verification are desirable for a design such as this one. This design is quite small, yet it allows for substantial concurrency and depends on the relative timing delays of its components. We hope that our work will provide a realistic test case for the verification community.

Acknowledgements

We have benefited from discussions about arbiter design with many of our colleagues. Foremost among them was Charlie Molnar to whom we are deeply grateful for his enthusiasm and insight. We also appreciate discussions we have had with Roger Brockett, Bill Coates, and Ian Jones.

References

[1] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In A. Hu and M. Vardi, editors, *CAV 98: Computer-aided Verification*, Lecture Notes in Computer Science 1427, pages 521–525. Springer-Verlag, 1998.

[2] W. Belluomini and C. J. Myers. Verification of timed systems using posets. In *Proceedings of the Tenth Conference on Computer Aided Verification*. Springer, June 1998.

[3] S. M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, Computer Science Department, California Institute of Technology, Pasadena, CA, Jan. 1991. Tech. Report Caltech-CS-TR-91–01.

[4] T. Chaney and C. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, Apr. 1973.

[5] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, Apr. 1997.

[6] M. R. Greenstreet and P. Cahoon. How fast will the flip flop? In *Proceedings of the 1994 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 77–86, Salt Lake City, Nov. 1994. IEEE Computer Society Press.

[7] M. R. Greenstreet and T. Ono-Tesfaye. Verifying a self-timed divider. In *Proceedings of the 1999 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, 1999. accepted for publication.

[8] M. R. Greenstreet and K. Steiglitz. Bubbles can make self-timed pipelines fast. *Journal of VLSI and Signal Processing*, 2(3):139–148, Nov. 1990.

[9] J. H. Hohl, W. R. Larsen, and L. C. Schooley. Prediction of error probabilities for integrated digital synchronizers. *IEEE Journal of Solid-State Circuits*, SC-19(2):236–244, Apr. 1984.

[10] J. U. Horstmann, H. W. Eichel, and R. L. Coats. Metatstability behavior of cmos asic flip-flops in theory and test. *IEEE Journal of Solid-State Circuits*, 24(1):146–157, Feb. 1989.

[11] H. Hulgaard, S. M. Burns, T. Amon, and G. Borriello. An algorithm for exact bounds on the time separation of events in concurrent systems. *IEEE Transactions on Computers*, 44(11):1306–1317, Nov. 1995.

[12] M. Hurtado. *Structure and Performance of Asymptotically Bistable Dynamical Systems*. PhD thesis, Sever Institute, Washington University, Saint Louis, MO, 1975.

[13] D. J. Kinniment, B. Gao, A. V. Yakovlev, and F. Xia. Toward asynchronous A-D conversion. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–215, 1998.

[14] L. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, Feb. 1981.

[15] A. J. Martin, S. M. Burns, et al. The design of an asynchronous microprocessor. In *Proceedings of the Conference on Advanced Research in VLSI*, Caltech, 1989.

[16] M. Mendler and T. Stroup. Newtonian arbiters cannot be proven correct. *Formal Methods in System Design*, 3(3):233–257, Dec. 1993.

[17] I. Mitchell and M. Greenstreet. Proving newtonian arbiters correct, almost surely. In *Proceedings of the Third Workshop on Designing Correct Circuits*, Båstad, Sweden, Sept. 1996.

[18] C. Molnar.

[19] C. E. Molnar, I. W. Jones, B. Coates, and J. Lexau. A FIFO ring oscillator performance experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 279–289. IEEE Computer Society Press, Apr. 1997.

[20] P. B. Pang and M. R. Greenstreet. Self-timed meshes are faster than synchronous. In *Proceedings of the 1997 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 30–39. IEEE, Apr. 1997.

[21] J. N. Seizovic. Pipeline synchronization. In *Proceedings of the 1994 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96. IEEE Computer Society Press, 1994.

[22] J. Sparsø and J. Staunstrup. Delay-insensitive multi-ring structures. *INTEGRATION*, 15(3):313–340, Oct. 1993.

[23] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design and Test*, 11(3):48–59, Fall 1994.

[24] S. Taşiran and R. K. Brayton. STARI: A case study in compositional and hierarchical timing verification. In *Proceedings of the Ninth Conference on Computer Aided Verification*, pages 191–201, Haifa, Israel, June 1997. Springer. LNCS 1254.

[25] L. Thiele. On the analysis and optimization of self-timed processor arrays. *INTEGRATION*, 12(2):167–187, Dec. 1991.

[26] J. T. Udding. *Classification and Composition of Delay-Insensitive Circuits*. PhD thesis, Eindhoven University of Technology, 1984.

[27] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, May 1991.

[28] T. E. Williams. Analyzing and improving latency and throughput in self-timed pipelines and rings. In *TAU 1992 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton, NJ, Mar. 1992.

[29] A. Xie and P. A. Beerel. Symbolic techniques for performance analysis of timed systems based on average time separation of events. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 64–75. IEEE Computer Society Press, Apr. 1997.

[30] A. Xie and P. A. Beerel. Accelerating Markovian analysis of asynchronous systems using string-based state compression. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 247–260, 1998.

A SPICE description of the arbiter

```
*********************************************
*
* the rlatch subcircuit
```

```
*
.SUBCKT rlatch set clr set2 ny nresB nres

Mnreset_y ny nres 0 0 mnch l=lmin w=1.6u

x1 set  set1 inv_rs
x2 set1 set2 inv_small
x3 set2 set3 inv_small
x4 set3 set4 inv_small

Mpy1 nvdd set4 n1 nvdd mpch l=lmin w='3*wp'
Mpy2 n1 set1 ny  nvdd mpch l=lmin w='3*wp'

x5 ny keeper
x6 clr clr1 inv_small
x7 clr1 clr2 inv_small
x8 clr2 clr3 inv_small

Mny1 ny clr n2 0 mnch l=lmin w='3*wn'
Mny2 n2 clr3 0 0 mnch l=lmin w='3*wn'

.ENDS rlatch

*********************************************
*
* the mutexHalf subcircuit
*
.SUBCKT mutexHalf nr nr2 ng ngB ny n_qB_o n_qB nqB g_o

Mp1 nvdd nr2 n_qB_o nvdd mpch l='4*lmin' w='1.6u'
Mp2 nvdd n_qB_o n_qB nvdd mpch l=lmin w='2*wp'
Mp3 nvdd n_qB_o nqB nvdd mpch l=lmin w=1.6u
Mp4 nvdd nr2 nqB nvdd mpch l='4*lmin' w=1.6u

Mp5 nvdd ngB n_qB nvdd mpch l='2*lmin' w='2.5*wp'
Mp6 nvdd ngB nqB nvdd mpch l='2*lmin' w='2.5*wp'

x1 ng nr2 nz nor

Mn3 n_qB nr n2 0 mnch l=lmin w='4*wn'
Mn2 n2 nz n3 0 mnch l=lmin w='4*wn'
Mn4 n_qB ny n3 0 mnch l=lmin w='4*wn'
Mn1 n3 n_qB_o 0 0 mnch l=lmin w='6*wn'
Mn11 n3 g_o 0 0 mnch l=lmin w='3*wn'

Mn5 n_qB n_qB_o nqB 0 mnch l=lmin w='4*wn'

.ENDS mutexHalf

*********************************************
*
* the nor subcircuit
*
.SUBCKT nor na nb ny

* the p-channel transistors
Mp1a nvdd  na n1a nvdd mpch l=lmin w='3*wp'
Mp2a n1a  nb ny nvdd mpch l=lmin w='3*wp'
Mp1b nvdd  nb n1b nvdd mpch l=lmin w='3*wp'
Mp2b n1b  na ny nvdd mpch l=lmin w='3*wp'

* the n-channel transistors
Mn3 ny na 0 0 mnch l=lmin w='2*wn'
Mn4 ny nb 0 0 mnch l=lmin w='2*wn'

.ENDS nor
*********************************************
*
* the dlatch subcircuit
*
.SUBCKT dlatch s1 s2 s1_1 s2_1 clr1 clr2 ny nres

Mnreset_y ny nres 0 0 mnch l=lmin w=1.6u

x0 s1 s1_1 inv_gb
x1 s1_1 s1_2 inv_small
x2 s1_2 s1_3 inv
x3 s1_3 s1_4 inv_big

x4 s2 s2_1 inv_gb
x5 s2_1 s2_2 inv_small
x6 s2_2 s2_3 inv
x7 s2_3 s2_4 inv_big

Mpy1_1 nvdd s2_4 n1 nvdd mpch l=lmin w='3*wp'
Mpy1_2 n1 s2_1 ny nvdd mpch l=lmin w='3*wp'

Mpy2_1 nvdd s1_4 n2 nvdd mpch l=lmin w='3*wp'
Mpy2_2 n2 s1_1 ny nvdd mpch l=lmin w='3*wp'

x8 ny keeper

x9 clr1 clr1_1 inv_small
x10 clr1_1 clr1_2 inv
x11 clr1_2 clr1_3 inv_big
Mny1 ny clr1 n3 0 mnch l=lmin w='6*wn'
Mny2 n3 clr1_3 0 0 mnch l=lmin w='4*wn'

x29 clr2 clr2_1 inv_small
x20 clr2_1 clr2_2 inv
x21 clr2_2 clr2_3 inv_big
Mny21 ny clr2 n3 0 mnch l=lmin w='6*wn'
Mny22 n3 clr2_3 0 0 mnch l=lmin w='4*wn'

.ENDS dlatch

*********************************************
*
* the arbiter subcircuit
*

.SUBCKT arbiter req1 req2 d1 d2 g1 g2 resB res

*the input-latches
x1a req1 g1 set2_1 y1 resB res rlatch
x2a req2 g2 set2_2 y2 resB res rlatch
```

```
*the new mutexHalf gate
x1b req1 set2_1 g1 g1B y1 qB_2 qB_1 qB1 g2 mutexHalf
x2b req2 set2_2 g2 g2B y2 qB_1 qB_2 qB2 g1 mutexHalf

*nor subcircuit
x1c qB1 f g1  nor
x2c qB2 f g2  nor

*the done-latch
x1d g1 g2 g1B g2B d1 d2 f res dlatch

.END
```