Real-Time Merging

Mark R. Greenstreet,*

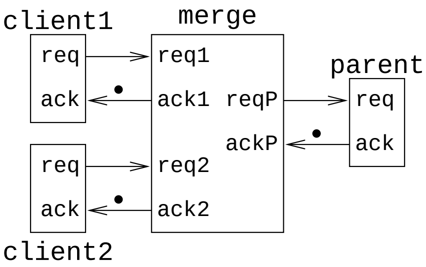


Figure 1. A Merge Element

Abstract

A merge element combines two, concurrent, handshake streams. For every request received from a client, a merge element may send a request to its parent, and for each acknowledgement received from its parent, the merge element may send an acknowledgement to a client. We show that that a merge-element can provide bounded time response if its parent also has bounded time response. We present two new implementations of a merge: one that uses an arbiter, and one that uses Schmitt triggers but no arbiters. Based on these designs, we explore a class of concurrent computations that can be performed in guaranteed bounded time, and we raise some new questions about what is possible in asynchronous design.

Keywords: arbitration, bounded-time, merging, metastability, real-time, receptive mixer.

1 Introduction

Consider the problem of merging two streams of handshakes. Figure 1 depicts a merge element connected to two clients and its parent. For simplicity, we assume transition signaling [16] (i.e. two-phase handshakes). The acknowledge input of an active port is marked with a dot to indicate that the first request event can be generated without first receiving an acknowledgement. The behavior of the merge is defined by the prefix closure of the rules below for traces involving the merge:

$$
\begin{array}{lll}
\text{Initially:} & & \\
 & (\#(\mathtt{req1}) = 0) \wedge (\#(\mathtt{ack1}) = 0) & (1.1)\\
\wedge & (\#(\mathtt{req2}) = 0) \wedge (\#(\mathtt{ack2}) = 0) & (1.2)\\
\wedge & (\#(\mathtt{reqP}) = 0) \wedge (\#(\mathtt{ackP}) = 0) & (1.3)\\
\mathit{Assume}: & & \\
 & \#(\mathtt{req1}) \leq \#(\mathtt{ack1}) + 1 & (1.4)\\
\wedge & \#(\mathtt{req2}) \leq \#(\mathtt{ack2}) + 1 & (1.5)\\
\wedge & \#(\mathtt{ackP}) \leq \#(\mathtt{reqP}) & (1.6)\\
\mathit{Guarantee}: & & \\
\wedge & \#(\mathtt{ack1}) \leq \#(\mathtt{req1}) & (1.7)\\
\wedge & \#(\mathtt{ack2}) \leq \#(\mathtt{req2}) & (1.8)\\
\wedge & \#(\mathtt{reqP}) \leq \#(\mathtt{ackP}) + 1 & (1.9)\\
\wedge & \#(\mathtt{reqP}) \leq \#(\mathtt{req1}) + \#(\mathtt{req2}) & (1.10)\\
\wedge & \#(\mathtt{ack1}) + \#(\mathtt{ack2}) \leq \#(\mathtt{ackP}) & (1.11)
\end{array}
$$

We write $\#(\mathtt{x})$ to denote the number of events that have occurred on $\mathtt{x}$. The three assumptions about the environment, equations 1.4–1.6, state that the behavior of the merge is only defined if the clients and parent observe the handshaking protocol. Equations 1.7–1.9 state that the merge element observes the handshake protocols. Equation 1.10 states that the merge can only make requests to its parent in response to requests from the clients. Finally, equation 1.11 states that the merge can only acknowledge requests from its clients after it has received acknowledgements from its parent.

1.1 Merge elements and arbiters

The merge element is closely related to the arbiter. As shown in figure 2, an arbiter can be constructed from a merge and a C-element.[1] The figure shows an RGD arbiter: client 1 (resp. 2) makes a request by asserting an event on `r1` (resp. `r2`). The arbiter issues

*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

[1] Thanks to Jo Ebergen for sketching this simple and elegant implementation on a napkin at Async98.

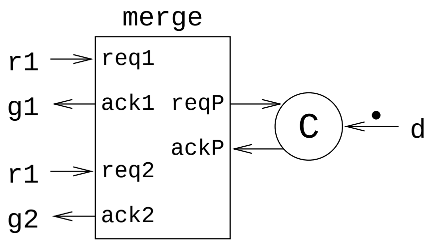


Figure 2. Constructing an arbiter from a merge

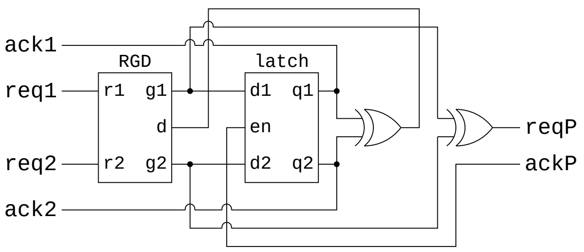


Figure 3. Constructing a merge from an arbiter

a grant to client 1 (resp. 2) by asserting an event on `g1` (resp. `g2`). When the client is done with the privilege, it releases it by asserting an event on `d`. The `C` element ensures grant events must alternate with done events; thus, at most one client is ever granted at a time.

Conversely, an arbiter can be used to construct a merge element. Figure 3 shows the construction from [17] that uses an RGD arbiter, a latch, two join elements (XORs, for transition signaling), and four forks. In this design, the RGD serializes concurrent requests from the two clients, the join connected to the grant outputs of the RGD combines these requests into a single stream, and the latch forwards acknowledgements from the parent back to the appropriate client. The join connected to the `q1` and `q2` outputs of the latch provides the done signal that allows the RGD to process the next request. In [17], this circuit is called a "receptive mixer."

1.2 Varieties of liveness and receptiveness

The properties specified in equations 1.1 – 1.11 describe safety properties of the merge. They do not require any form of liveness. In particular, an implementation that does nothing satisfies this specification, but such an implementation is probably unacceptable. We consider the following two forms of liveness:

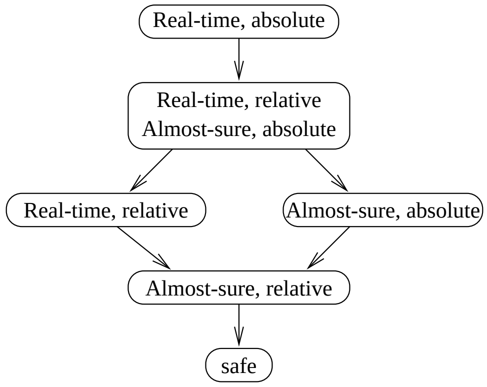


Figure 4. Varieties of liveness

Real-time liveness: A design is real-time live if there exists a real constant $\delta_{\max}$ such that if at some time, t_0, its specification allows an action to be taken, then some action is taken before time $t_0 + \delta_{\max}$.

Almost-sure liveness: A design is live, almost-surely [9] if at some time, t_0 its specification allows an action to be taken, then

$$\lim_{t \to \infty} P\{\text{action taken in } [t_0, t)\} = 1$$

We also consider relative liveness: for example, a design may be real-time live if and only if its environment is real-time live at one or more of its interfaces.

Real-time and almost-sure liveness in their absolute and relative versions allow us to describe six forms of liveness as depicted in figure 4. Arrows in the diagram denote implication; e.g., absolute real-time liveness implies all of the other forms, and relative real-time liveness implies relative almost-sure liveness. Furthermore, one can describe a merge as being receptive or non-receptive (see [17]). A receptive merge can accept concurrent requests from its clients; the environment of a non-receptive merge must ensure that only one client has a pending request at any time. Given the six forms of liveness and two forms of receptiveness, one could coin twelve names for variations on the merge element. We will avoid this quagmire, and refer to all of the variants as merge elements, explicitly stating the liveness guarantees provided by each design. It is straightforward to construct an absolute real-time, non-receptive merge (called a non-receptive mixer in [17]). In the remainder of this paper, we will only consider receptive designs.

It is impossible to implement an arbiter that is real-time or relatively real time live [4, 6, 8] due to metastability (see section 3. however, almost-sure liveness is achievable. For example, the design in figure 3 is a receptive merge with absolute almost-sure liveness.

1.3 Contributions

In this paper, we show that a merge-element can be implemented that is real-time live if its parent is real-time live. Given the close relationship between merge-elements and arbiters, this may seem surprising. The key difference between a merge-element and an arbiter can be seen by considering their allowed behaviors in the presence of concurrent requests from their two clients. Under these conditions, the arbiter may enter a metastable state that can persist arbitrarily long. This is why only almost-sure liveness can be achieved for the arbiter. Under the same conditions, the merge-element may make a request to its parent and receive an acknowledgement. Rather than deciding which client to acknowledge, the merge-element may proceed to make a *second* request to its parent; when this request is acknowledged, the merge element can then grant *both* clients. This is relative real-time liveness: if the parent is not real-time live, then no such guarantee can be made by the merge element.

Section 2 presents a simple merge that is real-time live if its parent is real-time live. This design does not use an arbiter and it is not almost-sure live unless its parent is. The design of section 2 is uses a Schmitt-trigger, and section 3 compares the dynamics of Schmitt triggers to those of mutexes. Section 4 presents a merge that uses a mutex and is real-time live if its parent is real-time live and absolutely almost-sure live. Section 5 applies the ideas in these merge designs to operand garnering for a Sproull Counterflow Pipeline Processor [14]. Finally, section 6 explores some of the questions of what is real-time computable and what elements are required.

2 Implementing a Merge Element with Schmitt Triggers

Figure 5 shows a design of a merge that follows directly from equations 1.1 – 1.11. This section gives an intuitive explanation of how the design works. This particular design is not attractive for a practical implementation because of its many timing dependencies. It is presented to demonstrate that a receptive merge that is real-time live relative to its parent can be implemented without using an arbiter. In appendix A, the design is formalized as a timed transition system and verified by establishing an invariant for the system.

The heart of the design is the "count-and-compare" unit (labeled `a+b cmp c` in the figure). This module

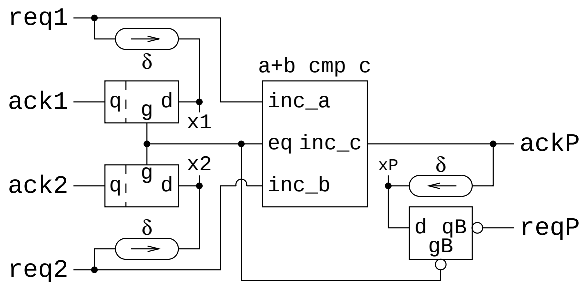


Figure 5. Implementing a Merge Element with Schmitt Triggers

has three counters, `a`, `b`, and `c`. When an event is received on the `inc_a` input, the `a` counter is incremented, likewise for the `inc_b` and `inc_c` inputs and `b` and `c` counters. The boxes driving `ack1` and `ack2` are transparent latches: when the `g` input is true, the latch copies the value of its `d` input to its `q` output. When the `g` input is false, `q` holds its value. The dashed line is included to indicate that `q` is driven by a Schmitt-trigger buffer. The box driving `reqP` is an inverting transparent latch with inverted enable. The objects labeled δ are buffers with a minimum delay of δ time units. We write δ_{CC} to denote the maximum delay from an `inc_a`, `inc_b`, or `inc_c` input of the count-and-compare unit to the `eq` output, and we require that $\delta_{CC} < \delta$. The designs of the Schmitt-trigger latch and count-and-compare unit are described in more detail in sections 2.1 and 2.2 respectively.

When $\mathtt{a} + \mathtt{b} > \mathtt{c}$, there are outstanding requests from the clients, and the merge may issue a request to its parent. This is accomplished by enabling the latch that drives `reqP` so that it can set `reqP` to the inverse of `ackP`. When a new acknowledgement is received on `ackP`, `eq` is set before `xP` because $\delta_{CC} < \delta$; thus, no spurious requests are made. If $\mathtt{a} + \mathtt{b} = \mathtt{c}$, then the merge has received an acknowledgement for every client request. This enables the transparent latches driving `ack1` and `ack2` so they can set `ack1` to the value of `req1` and `ack2` to the value of `req2` thereby acknowledging any pending requests. The requirement that $\delta_{CC} < \delta$ ensures that, after a client request, the `eq` signal is updated before the `x1` or `x2` signal changes. This ensures that clients do not receive spurious acknowledgements.

The design has a hazard in that the `eq` signal may go high for an arbitrarily short interval. The other actions are hazard free. For example, following a request from client 1, a hazard on `eq` occurs if an acknowledgement is received from the parent at roughly the same as a client makes a request, then `eq` might or might not be set to true depending on whether or not the `c` counter

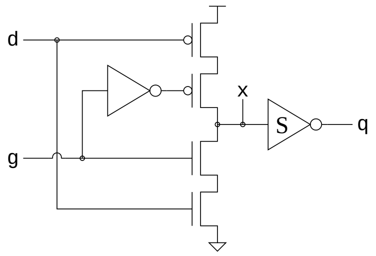


Figure 6. A Latch with Schmitt trigger buffering

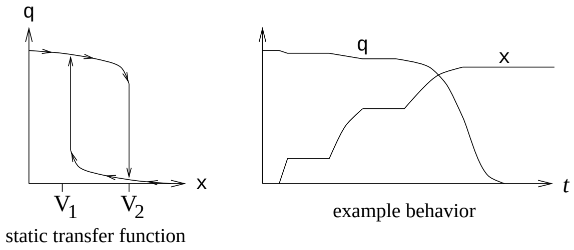


Figure 7. Schmitt trigger characteristics

is incremented before the `b` counter. This produces a glitch on the `g` signal of the latch that drives `ack1`. Signal `x1` makes exactly one transition for each handshake of client 1. Before the value of `req1` propagates to `x1`, the latch output remains unchanged during glitches on `g`. After `x1` = `req1`, the latch is enabled to drive its output to this value each time `g` is high. Section 2.1 describes a latch implementation that does not produce output glitches in spite of glitches on `g`.

An ordinary latch can be used for the latch that outputs `reqP`. If a marginal high pulse occurs on `eq`, then `reqP` $\neq$ `xP` holds on the rising edge of the pulse. To see this, note that the only event that can set `eq` to true is an increment of the `c` counter. This happens in response to an `ackP` event which means. When a request is outstanding, `reqP` $\neq$ `xP`. When an `ackP` event is received, `eq` is updated within δ_{CC} time units and `xP` remains unchanged for at least δ time units. By the constraint that $\delta_{CC} < \delta$, `reqP` $\neq$ `xP` at the rising edge of `eq`. Thus, there are no hazards for the latch that outputs `reqP`.

Successive requests from a client must be separated by at least δ time units (the delay from `req1` to `x1` or from `req2` to `x2`). When a new request is received, the `a` or `b` counter is incremented within δ_{CC} time units. By the requirement that $\delta_{CC} < \delta$, these counters are hazard-free.

2.1 The Schmitt-trigger latch

Figure 6 shows the Schmitt-trigger latch circuit that produces `ack1` and `ack2`. First consider the operation of the latch when its inputs are free of glitches. When `g` is high, the internal node, `x`, is driven to the logical negation of the `d` input. Then, the Schmitt-trigger inverter drives `q` to the same value as `d`. When `g` is high, `x` retains its value, and `q` does as well.

Now consider what occurs if a short high pulse occurs on `g`. If the current request has not yet propagated to `d`, then `d` and `x` have complementary values, and a pulse on `g` leaves `x` unchanged. When `d` takes on the value of a request, then whenever `g` is true, the voltage of `x` will change toward the complement of `d`, and when `g` is false, `x` will hold its voltage. Thus, `x` changes monotonically. Once `x` reaches the logical negation of `d`, it will hold that value during any future pulses on `g`. The `d` input will not change again until after the request that led to this value of `d` has been acknowledged.

Figure 7 shows the transfer characteristics of a typical Schmitt trigger inverter. The key feature of a Schmitt trigger is the hysteresis in its transfer function. When the input makes a low-to-high transition, the output follows the upper curve. When the input reaches the upper threshold voltage, V_2, the Schmitt trigger transitions to the lower curve. This transition is effected by a positive feedback mechanism: if the input makes a monotonic transition across the threshold, the output will complete the transition even if there is no further change in the input. This is what allows the Schmitt trigger to convert the monotonic transition on `x` to a sharply defined output transition on `g`. CMOS implementations of Schmitt trigger circuits are presented in [3].

If a short `eq` pulse brings `x` just barely past the Schmitt trigger threshold, the Schmitt trigger may take arbitrarily long to produce an output transition on `q`. Thus, an output event may occur while the `g` input is false. Fortunately, the merge implements a delay-insensitive interface. Thus, this delay does not affect the correctness of the design. Furthermore, if the parent responds in bounded time, then an ungranted request will lead to a state where `eq` is true after a bounded delay.

2.2 The count-and-compare unit

There are two issues that must be addressed in the design of the count and compare unit. First, real counters have only finite range. As shown by equation 3 in appendix A, $\texttt{a} + \texttt{b} - \texttt{c}$ is always in the range $0 \ldots 2$. Thus, modulo-K arithmetic can be used in the count-and-compare unit for any $k \geq 3$. Our design uses

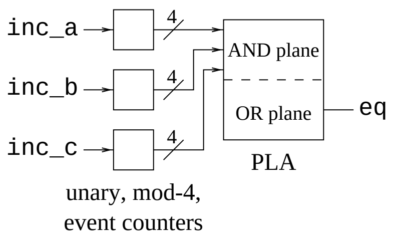


Figure 8. Count-and-compare unit

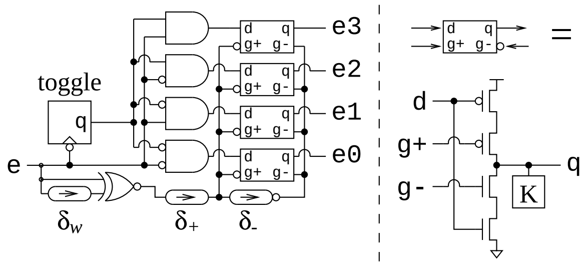


Figure 9. A unary, mod-4, event counter

modulo-4 counters.

The other issue is generating the eq signal without glitches when the counts change. Figure 8 shows a design that avoids such glitches. Each unary, mod-4, event counter generates one of its four output wires high to indicate the total number of events that it has received from its input. The counters are designed so that after each input event the output for the new count goes high before the output for the old count goes low. These two events are separated by at least the maximum delay of the PLA. The counter outputs are glitch free. The design of such a counter is presented later in this section.

The PLA computes the eq signal based on the counter outputs. Each minterm in the PLA has exactly one input from each counter, and none of the inputs are inverted within the PLA. Let $P(a, b, c)$ be some predicate on the values of counters a, b, and c, and consider what happens when one or more counters are incremented. There are no low-going glitches on the output of the PLA by the monotonicity of the PLA and the property that the counters output rising events before falling ones. A high-going glitch can only occur if for some ordering of the a, b, and c events, P was transiently satisfied. For the design from figure 5, these are exactly the requirements for the eq signal. We will use these properties again in section 4 for a variation on this count-and-compare design.

Figure 9 shows the design of an event counter. The output of the toggle flip-flop transitions once for every falling edge of the input. Thus, the value of the input is the least-significant bit of the number of events mod-4, and the output of the counter is the most-significant bit. The four AND gates decode this to a unary representation. The δ_w delay element and XNOR gate produce a low going pulse for each input, δ_w sets the width of this pulse. The delay, δ_+ is chosen to be greater than the longest delay through the toggle and AND gates. Thus, when the pulse arrives at the g+ input of the latches, the outputs of the AND gates are stable. This ensures glitch free rising events on the counter's outputs. The delay δ_- is chosen to be greater than the longest delay through the PLA in figure 8. Thus, rising output events from the counter propagate through the PLA before falling events arrive.

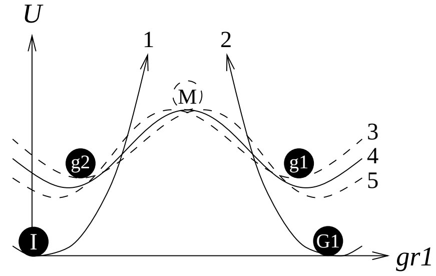
2.3 Liveness

The design shown in figure 5 is real-time live relative to its parent. First, note that if the merge receives a single request, it will respond in bounded time. The delay is bounded by the sum of twice the delay of the count-and-compare unit, the delay of the parent, twice the delay of a transparent latch, and δ. Each of these delays is bounded, therefore the total delay is bounded as well.

Now consider what happens if client 1 makes a request and then client 2 makes a request before client 1 receives an acknowledgement. The request from client 2 may disable the latch that produces ack1 before an acknowledgement is output. In this case, the merge will make a second request to its parent which again will be acknowledged in bounded time. No new client requests will be received until at least one client is granted. Thus, eq will be set to true within bounded time of the first request, and one of the clients will be acknowledged within bounded time.

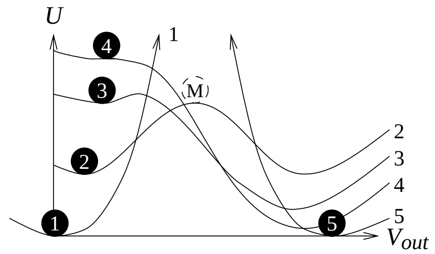
Although the definition of real-time liveness in section 1 did not require fairness, the design presented here can be made to be fair. Fairness is guaranteed, for example, if the minimum client delay is greater than the maximum delay of an enabled latch. For simplicity, we have omitted these delays from our models.

The design in this section does not satisfy any absolute liveness properties. For example, consider a scenario where both clients make requests in rapid succession. Both the a and b counters are incremented, and a request is issued to the parent. When the parent responds with an acknowledgement, the c counter is incremented. Now, $\mathtt{a}+\mathtt{b}-\mathtt{c} = 1$. The merge will output another request to the parent. If the parent does not reply to this request, then the merge will never reach a state where $\mathtt{a}+\mathtt{b}-\mathtt{c} = 0$. Under these conditions, the merge element will never produce an acknowledgement



1 :	$req_1 = V_l,\ req_2 = V_l$	2 :	$req_1 = V_h,\ req_2 = V_l$
3 :	$req_1 = V_h - \epsilon,\ req_2 = V_h$	4 :	$req_1 = V_h,\ req_2 = V_h$
5 :	$req_1 = V_h + \epsilon,\ req_2 = V_h$		

Figure 10. Arbiter dynamics



1 :	$V_{\text{in}} = V_h$	2 :	$V_{\text{in}} = (V_a + V_b)/2$
3 :	$V_{\text{in}} = V_a + \epsilon$	4 :	$V_{\text{in}} = V_a$
5 :	$V_{\text{in}} = V_l$		

Figure 11. Schmitt trigger dynamics

for either client. In other words, this design should not be used in an environment where the parent is not guaranteed to respond to requests; otherwise, deadlock may occur. For example, if this merge is used in the arbiter construction from figure 2, deadlock can occur when both clients make concurrent requests. Section 4 presents a merge that is both relatively real-time live and absolutely almost-sure live.

3 Schmitt Triggers and Arbiters

The merge design presented in the previous section is distinguished by the absence of an arbiter. Instead, Schmitt trigger latches ensure that the merge element's outputs are well-behaved, even in the presence of concurrent inputs. This section compares the dynamics of arbiters and Schmitt triggers.

Figure 10 illustrates the dynamics of a simple arbiter. A "potential"[2], U, is plotted as a function of the grant 1 output voltage for various input conditions. Trajectories head downhill: local minima correspond to stable equilibria, and local maxima correspond to unstable equilibria (i.e. the metastable point).

Curve 1 depicts operation with both requests at the logical-low voltage, V_l. There is a single equilibrium point for *gr1* which is near V_l. The "ball" labeled I marks the arbiter resting at this stable point. The upward arrow at the right end of curve 1 indicates that the "potential" continuous to grow for increasing values of `gr1`; it does not indicate the direction of the trajectory. Curve 2 depicts operation with only client 1 making a request (i.e. $req_1 = V_h$, the logical-high voltage). Here, the stable point for *gr1* is near V_h, and this is indicated by the ball labeled G1. If both clients make requests, the arbiter exhibits bistable behavior: it has stable equilibria for *gr1* near V_l and V_h and a saddle point between these two. If a trajectory is near the saddle point, we say that the arbiter is metastable, and it may take arbitrarily long before the ball rolls down to one of the stable points. This situation is depicted by curves 3–5.

Small changes of the inputs do not remove the problem of metastability. If both req_1 and req_2 are at V_h (i.e. curve 4) and the trajectory is near this maxima point for this curve, then increasing req_1 to $V_h + \epsilon$ can move the arbiter to curve 5, and the trajectory will head to the equilibrium marked g2. However, if the ball had already rolled slightly towards g1 when req_1 increases, this can move the metastable point to the trajectory. It is well-known that there is no way that an arbiter can be guaranteed to respond in bounded time when presented with concurrent requests [2, 4, 13, 6, 8].

Like an arbiter, a Schmitt trigger can exhibit bistable behavior (the upper and lower curves in figure 7). Unlike an arbiter, a Schmitt trigger has only one input. Figure 11 illustrates the dynamics of a Schmitt-trigger inverter. Consider operation that starts with the input high, and the output, V_{out} near V_l (with $V_l < V_a$). This situation is depicted by curve 1 with the output near the equilibrium point and marked by ball 1.

If the input falls monotonically to $(V_a + V_b)/2$, then the Schmitt trigger has the dynamics depicted by curve 2. There are two stable equilibria and one metastable point. The trajectory that started near V_l follows the left, stable equilibrium as the input changes. In the figure, the left equilibrium is marked by ball 2, and the metastable equilibrium is marked by the ball labeled M. When the input is just above V_a, there are two stable equilibria – the dip for the left minima has become quite small (curve 3). The left minima disappears when $V_{\text{in}} = V_a$ (curve 4). At this point, the potential has an inflection. This is an unstable equilibrium, from which trajectories can only exit to the right. As depicted by

[2] This "potential" could be formalized using Lyapunov functions. However, the comparison presented here is intended to aid intuition. A highly formal treatment is unnecessary and would probably be counter-productive.

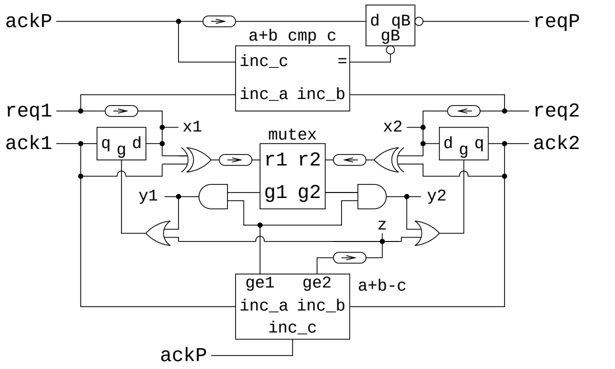


Figure 12. Implementing a Merge Element with Mutex

curve 5, the Schmitt trigger has a single equilibrium point near V_h when the input is below V_a.

Although a Schmitt trigger can exhibit metastable behavior, most applications require that the input change monotonically after crossing a switching threshold. Under these conditions, metastability will never occur. In fact, once the output begins to make a transition, further progress of the input will only make the output transition faster. Thus, a Schmitt trigger can guarantee bounded rise and fall times for output transitions if the input changes monotonically. As noted in the section 2.1, the inputs to the Schmitt triggers in the latches make monotonic transitions. This ensures that the `ack1` and `ack2` outputs make sharp transitions that swing completely from V_l to V_h.

4 Merging with a Mutex

Figure 12 shows a merge that uses a mutex but no Schmitt triggers. Like the design from figure 5, this design uses one count-and-compare unit to determine when requests should be issued to the parent. A second count-and-compare unit determines how many acknowledgements are available for issuing to the clients. The `ge1` ("greater than or equal to one") output of this count-and-compare unit is set true if at least one acknowledgement from the parent has not yet been issued to a client. Likewise, the `ge2` output is set true if at least two acknowledgements have been received from the parent but not sent further to the clients. The mutex is a level mutex (i.e. four-phase signaling), and the XOR gates convert the transition signals of the clients to four-phase signals for the mutex.

The design in figure 12 is presented to show that a receptive merge can be implemented that is absolutely almost-sure live and relatively real-time live. Like the circuit in figure 5, this design would be difficult to implement in practice because of its many delay constraints. The minimum delay through the delay element connected to `ackP` must be greater than the maximum delay through the upper count-and-compare unit. The minimum delay through the delay element driving `mutex.r1` must ensure that path from `ack1` via the mutex to the AND gate for `ack2` is longer than the path from `ack1` through the count-and-compare unit to `ge1`. Likewise for the delay element driving `mutex.r2`. The minimum delay through the delay element connected to `ge2` must ensure that the path from the `ack1` or `ack2` through the count-and-compare unit to the `z` must have a greater delay than the path from `ack1` or `ack2` through the mutex to the upper inputs of the OR gates. The minimum delay through the delay element connected to `req1` or `req2` must be greater than the maximum delay from `ack1` or `ack2` through the count-and-compare unit to the `g` inputs of the latches that produce `ack1` and `ack2`. The internal mutex must be absolutely, almost-surely live. Furthermore, if an ungranted request is removed from the mutex, the mutex must produce valid outputs again within a bounded amount of time. Many common mutex implementations (such as cross-coupled NAND gates with a metastability filter [7]) satisfy these requirements.

Client requests are forwarded to the parent through the upper count-and-compare unit. They are also forwarded to the mutex. If there is one outstanding acknowledgement from the parent (i.e. `ge1` is true and `ge2` is false), then this acknowledgement is forwarded through the AND gate and transparent latch to the client that is chosen by the mutex. On the other hand, if there are two outstanding acknowledgements, then the OR gates enable both transparent latches and both clients are granted.

The delay constraints ensure the following ordering of events when client 1 is granted: First, a transition occurs on `ack1`. Second, `ge1` goes low (if all parent acknowledgements have been consumed). Third, the internal mutex is released by this client. Fourth, the mutex is acquired by the other client (if it has a pending request). Fourth, `z` goes low (if it had been high). Only after all of these have occurred, can a subsequent request from client 1 propagate to `x1`. We can now show that the conditions for the output latches are satisfied.

The latches that generate the acknowledge output incur no hazards. The `ge1` signal is always high while a grant is being issued. Consider what happens if `ge2` changes while a grant is being issued. If `ge2` goes from false to true, `ge1` remains true, and if either latch had been enabled, it will remain enabled. If `ge2` goes false, this means that a client has been acknowledged; if the other client is still unacknowledged, it has already acquired the mutex.

Furthermore, no spurious acknowledgements are issued. First consider the case that the latch that drives `ack1` is enabled because `ge1` is high. The timing constraints ensure that `ge1` was updated after the previous client acknowledgement before the mutex was acquired. Thus, `ge1` is valid. Now consider the case that the latch that drives `ack1` is enabled because `ge2` is high. Then, either there are two parent acknowledgements available, or client 2 was just granted, and there is one remaining acknowledgement for client 1. In either case, the grant to client 1 will be legitimate.
The above arguments show that the merge is safe.

If this merge-element receives an acknowledgement from its parent and its internal mutex is hung in a metastable state, within a bounded amount of time, the merge will issue another request to its parent. If the parent is real-time live, then it will send a second acknowledgement; `ge2` will be set to true; and both clients will be granted. Thus, this design is real-time live relative to its parent.

This design is also absolutely almost surely live. The key concern here is the behavior of the merge when it receives a request from each client and only one acknowledgement from its parent. In this case, it grants a client after the internal mutex makes a decision, and the internal mutex was required to be absolutely almost surely live.

5 Garnering Without Arbitration

The Sproull CounterFlow Pipeline Processor (CFFP [14]) has been actively studied in the asynchronous community. This architecture consists of two pipelines: an instruction pipeline where instructions flow up through the functional units; and a result pipeline where completed results flow down. When an instruction passes a result that is destined for an operand register of the instruction, the value must be copied from the result pipeline to the instruction pipeline. This operation is called garnering. To ensure that every instruction has an opportunity to interact with every result, previous designs have used an arbiter to make sure that each stage boundary instructions and results are transferred one at at time. In addition to ensuring that each instruction has an opportunity to garner each result, this arbiter also determines the stage in which the garnering occurs. In this section, we show how to transfer the proper value without knowing the stage from which it came. This corresponds to the key idea in the merge: namely that an acknowledgement can be issued to a client without designating a particular parent acknowledgement as the source.

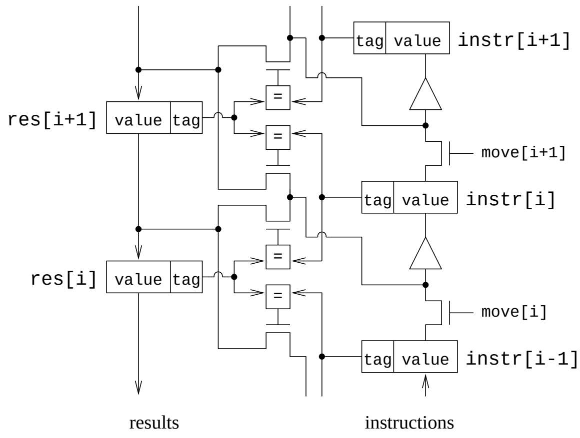


Figure 13. Garnering Without Arbitration

Figure 13 sketches our garnering technique for a highly simplified version of the CFPP. Each instruction stage can receive data from any of three sources: the previous instruction stage, the adjacent result stage, or the result stage one position higher in the pipeline. The transfer paths for data are indicated by pass transistors in the figure. These transistors along with the buffer that they drive form a multiplexed latch.

To transfer a data value to stage `i` from stage `i-1`, the `move[i]` signal is driven high. Tags are transferred with the data; the connections and control for the tags are omitted from the figure to focus on garnering. The boxes labeled `=` are comparators. If the tag for instruction matches the tag of the result in the corresponding result-pipeline stage, then the value is transferred. Furthermore, if the instruction tag matches the result tag in the stage one higher in the pipeline, the value is transferred. Each value register includes a valid bit to indicate whether or not a value has been garnered.

Now, assume that each instruction spends at least t_s time in each pipeline stage, and that each result spends at least t_s time in each result pipeline stage. Assume that it takes at most t_m time units to move an instruction between pipeline stages, or to move a result, with $t_m < t_s$. Let $t_g = (t_s - t_m)/2$. Each garnering operation has at least t_g time units to complete. To see this, consider the moment when a result first reaches the stage one above that of an instruction that needs it. This result will remain in that stage for at least t_s time units. The instruction will either remain in the stage one lower for t_g time units, in which case garnering will complete in the current stage, or the instruction will move to the next stage less than $t_s - t_g$ time units after the result arrived. In the second case, at least

t_g time units are available for the garnering after the instruction moves.

Consider what happens in the above scenario if the instruction moves up less than t_g time units after the result becomes visible. The instruction may have successfully garnered the result, or it may have a garbage value. In the latter case, the garbage value will be overwritten after the move, and in either case the instruction acquires the correct value. Conversely, consider what happens if the instruction moves more than t_g time units after the result becomes visible. In this case, the instruction moves to the next stage with the correct value. The garnering transistors in the next stage may be enabled for less than t_g time units. However, they are driving the input to the buffer to the same value that it already has, and no damage is done.

6 Implications

In sections 2 and 4 designs for merge elements have been presented that are real-time live relative to their parents. This allows us to answer some questions about what can be computed in bounded time, while raising some new questions. This section examines some of these issues.

6.1 What we know

It is possible to implement a merge element without using an arbiter.

The design of section 2 is an example of such a design.

It is possible to implement a merge element that is real-time live relative to its parent and absolutely, almost-surely live.

The design of section 4 is an example of such a design.

It is not possible to implement a merge element that is absolutely almost-surely live without using an arbiter.

Figure 2 shows an arbiter that is constructed from a merge element. If the merge element is absolutely almost-surely live, then the constructed arbiter will be as well. When such a merge is presented with a single acknowledge from its parent in response to two requests from its clients, it must choose which client to grant. The standard continuity arguments for arbiters (e.g.[4, 6, 8]) apply to this merge, which means that it must contain a component that can exhibit metastable behavior. Thus, an absolutely almost-surely live merge must contain an arbiter, a Q-flop [12] or equivalent.

A Schmitt-trigger does not provide the bistable functionality required for an arbiter if the transitions of its inputs are guaranteed to be monotonic after they cross a logic threshold. Although Schmitt-triggers are commonly used to implement "metastability" filters, the author is not aware of any design where a Schmitt trigger is used as the bistable element.

There exist a delay-insensitive decomposition of the merge element that is almost-surely live but not real-time live relative to its parent.

Figure 3 from [17] is an example of such a design.

It is not possible to implement a merge element using only gates and delay elements.

We note that all traditional, primitive, delay-insensitive building blocks except for the arbiter can be implemented using only gates and delay elements.

To see that gates and delay elements are insufficient to implement a merge, we assume that these components are monotonic in the arrival times and information content of their inputs. The first monotonicity condition means that if a gate or delay element produces output event E (i.e. a rising or falling transition) at time t in response to some set of input events, then if the input events are modified to make one or more of the inputs arrive earlier, then either E occurs at a time less than or equal to t, or the element produces an undefined output at time t. In other words, the gate or delay element cannot generate an output later in response to receiving an input earlier. The second monotonicity condition means that if some set of input events generates E at time t as above and the set is modified to make one or more inputs take on undefined values that had previously had defined values, then the the circuit generates either E or an undefined value at time t. In other words, changing the inputs of a gate from well-defined values to undefined values cannot change the output from an undefined value to a well-defined value.

We also assume that all gates and delay elements have some minimum delay. With this assumption, a simple induction argument shows that the two monotonicity conditions stated above hold for the entire network.

Now, we employ a classical mean-value argument as is common with arbiters. Consider a merge element that receives exactly one request from client 1, and this request is received at time t_1. Assume that the parent acknowledges its first request in bounded time and never responds to subsequent requests. At some time, the merge element must issue an acknowledgement to the client. Let t_2 be a time such that client 1 is acknowledged at time t_2. Let t_3 be some time such that $t_3 > t_2$. If the merge element is operated as above with the additional condition that it receives exactly one request from client 2, and this request is received at time t_3, then at time t_2, client 1 will be acknowledged. Now, there is some time $t_0 < t_1$ such that if client 2 makes are request at time t_0, then client 1 will never be granted. In particular, client 1 will not be granted at time t_2. Standard continuity arguments can be used to show that there is some time, t' between t_0 and t_3 such that at time t_2, the acknowledge signal for client 1 is midway through its transition. This means that by making the client 2 request occur earlier, the client 1 acknowledgement occurs later. As stated above, this is not possible with gates and delay elements with positive minimum delays that are monotonic with respect to event times and information content. Thus, it is not possible to build a merge only using such elements.

6.2 What we don't know

Is there a delay-insensitive decomposition of a merge-element that is real-time live relative to its parent?

To show such liveness, a delay model (e.g. bounded delay) will be required. However, the merge should be safe in the delay-insensitive model.

Do Schmitt triggers allow real-time computations that can't be performed with gates, delay elements, and arbiters?

Originally, merging seemed to be a candidate for such a computation, but the construction of figure 12 shows that merging can be performed without using Schmitt triggers and still satisfy real-time liveness relative to its parent. It seems that a careful definition of real-time liveness is needed along with a framework in which invariants of networks with and without Schmitt triggers can be expressed.

Can a value combining merge be implemented that is real-time live relative to its parent?

Unlike an arbiter, the merge does not assign an order to client events – clients may be acknowledged concurrently. This is the fundamental difference between a merge and an arbiter. For example, if both clients request concurrently, then two requests will be eventually forwarded to the parent, but we cannot necessarily identify which parent request came from which client. The "eager request" arbiter in [5] has a similar property for requests. However, a mutex in each node of their arbiter tree designates a client for each acknowledgement from the parent. In our design, we cannot necessarily identify which parent acknowledgement led to a particuar acknowledgement of a client. Thus, parent events do not imply an ordering of client events.

One could imagine encoding values with requests. If these values are combined by an associative and commutative operator, then the order of the operations doesn't matter. For example, one might design a tree that computes the sum of values encoded with requests or that finds the maximum of the values. If the values are encoded in unary, then their sum is easy to extract: it's the total number of handshakes performed with the parent. However, unary encodings are impractical for almost any application.

An open challenge is to design a real-time, relatively live merging adder that uses binary (or another compact) representation. Alternatively, one could prove that a such an adder can not be constructed. Either result would further illuminate what concurrent computations can be performed in bounded time and which can only be performed almost-surely.

Does bounded time merging have practical applications?

This paper has demonstrated that bounded time merging is possible by presenting two implementations. However, these designs are not very efficient. In particular, the overhead of the logic in these designs would dominate the time required for arbitration in all but an extremely small percentage of the cases. The CFPP example is also only proof-of-concept. On the other hand, all three designs show that unbounded metastability delays can be avoided in contexts where they previously seemed unavoidable. It is the authors hope that these observations will contribute to our understanding of concurrency and eventually contribute to practical designs.

7 Conclusions

This paper has explored designs for merge elements that combine two concurrent handshake streams into a single stream. Previously published designs have used arbiters in a way that the merge element could take arbitrarily long to perform an operation even if its clients and parent always respond in bounded time. This paper has presented two designs that make bounded time guarantees.

The first design is distinguished in that it uses Schmitt triggers and does not use an arbiter. This design will respond to all client requests in bounded time if its parent also responds in bounded time. However, this design has the annoying property that if its parent does not respond to all requests, the merge may receive an acknowledge from its parent that it never forwards to a client. The second design uses an arbiter. Like the first design, this merge responds to all client request in bounded time if its parent does the same. All grants from the parent are eventually forwarded to a client, perhaps after an arbitration.

Finally, we showed how the ideas that motivated these merge element designs can be used to implement operand garnering without arbitration in a simplified version of the Sproull Counterflow Pipeline.

These designs show that there is a class computations that involves combining concurrent streams but where arbitration is not required. Unlike arbitration, such computations can be performed in bounded time. Several properties of these computations were identified, and many new questions have been raised.

Acknowledgements

This problem was brought to my attention when Ivan Sutherland noted that "arbiters perform two functions: they merge streams of events, and they impose an order on these events." He asked if these two functions could be separated. This paper is a response to that question. I have also benefited from discussions with Jo Ebergen, Ian Jones, Alain Martin, Willem Mallon, and Nick Pippenger. The anonymous referees provided many helpful comments on an earlier draft of this paper.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. In J. de Bakker et al., editors, *Proceedings of the REX Workshop, "Real-Time: Theory in Practice"*. Springer, 1992. LNCS 600.

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

[3] L. A. Glasser and D. W. Dobberpuhl. *The Design and Analysis of VLSI Circuits*. Addison-Wesley, 1985.

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

[5] M. B. Josephs and J. T. Yantchev. CMOS design of the tree arbiter element. *IEEE Transactions on VLSI Systems*, 4(4):472–476, Dec. 1996.

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

[7] A. J. Martin. Programming in VLSI: From communicating processes to delay insensitive circuits. In C. Hoare, editor, *University of Texas Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1989.

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

[9] 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.

[10] T. Ono-Tesfaye, C. Kern, and M. R. Greenstreet. Verifying a self-timed divider. In *Proceedings of the 1998 International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 146–158, 1998.

[11] A. P. Ravn and J. Staunstrup. Synchronized Transitions. Technical Report DAIMI PM-219, Computer Science Department, Aarhus University, Århus, Denmark, Jan. 1987.

[12] F. U. Rosenberger, C. E. Molnar, et al. Q-Modules: internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9):1005–1018, Sept. 1988.

[13] C. L. Seitz. System timing. In *Introduction to VLSI Systems* (Carver Mead and Lynn Conway), chapter 7, pages 218–262. Addison Wesley, 1979.

[14] 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.

[15] J. Staunstrup and N. Mellergaard. Localized verification of modular designs. *Formal Methods in System Design*, 6:295–320, 1995.

[16] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. Turing Award lecture.

[17] T. Verhoeff. Encyclopedia of delay-insensitive circuits, 1998.

A Verification

Section 2 presented a simple design for a merge that showed that merging can be done without arbitration. The arguments in that section were intuitive and operational. This appendix presents a more formal derivation of the properties of this design.

We model the merge element and its clients and parent using the Synchronized Transitions notation [11]. A transition is written as $\ll G \rightarrow M \gg$ where G is a Boolean valued expression (the guard), and M is a multi-assignment. A transition is enabled in any state where its guard expression evaluates to true. Transitions may be combined with the asynchronous combinator, $\|$. Given a program of the form $t_1\|t_2\|\ldots\|t_n$, an execution consists of repeatedly choosing a transition, testing its guard, and performing the multi-assignment if the guard is satisfied. The order in which transitions are selected is unspecified.

For the merge element design, timing properties of the design must be modeled, and we use the same approach as presented in [10] which is based on Abadi and Lamport's "Old-Fashioned Recipe" [1]. In particular, we introduce a real-valued variable, τ, that represents the current time. By writing a transition as $\ll(\tau \geq X) \wedge G \rightarrow M\gg$, we can express the constraint that the transition cannot be performed before time X. We model time as a quantity controlled by the environment, and a "protocol" (adapted from [15]) to describe how time progresses. In particular, we state that time increases monotonically, and express upper bounds for delays of some actions.

Figure 14 shows the Synchronized Transitions program for the merge element of figure 5. The variables `#req1`, `#ack1`, `#req2`, `#ack2`, `#reqP`, and `#ackP` are auxiliary variables that keep track of the counts that appear in the merge element specification (see equation 1). The assume clause, $\delta > \delta_{\mathrm{CC}}$ states that the minimum delay of the delay elements driving `x1`, `x2`, and `xP` is greater than the maximum delay through the count-and-compare unit. The initially clause describes an initial state where no request or acknowledgements have yet occurred on any interface, the merge-element is in equilibrium, and both clients are enabled to make requests. The protocol section describes how time advances: the first clause states that time increases monotonically; and the last three clauses give upper bounds for delays of the count-and-compare unit. The transitions section models the behavior of the merge element and its environment.

To verify the merge, we use the invariant that is presented in figure 15. Although the invariant is as long as the program it describes, it has a relatively simple explanation. Clauses 7.1 through 7.3 describe the handshake loop for client 1 (i.e. signals `req1`, `x1`, and `ack1`); the clauses state that at most one component along this loop is enabled at any time. Likewise, clauses 7.4 through 7.9 describe the interface with the parent. Clauses 7.10 through 7.12 couple the state of the handshake loops to the values of the corresponding counters in the count-and-compare unit. Finally, clauses 7.13 through 7.15 are based on the observation that $\texttt{\#ackP} \leq \texttt{\#req1} + \texttt{\#req2} \leq \texttt{\#ackP} + 1$ and this observation is reflected by the property that $\texttt{c} \leq \texttt{a} + \texttt{b} \leq \texttt{c} + 1$. Clauses 7.13 through 7.15 give properties that hold in each of the handshake loops for each of the three possible values of $\texttt{a} + \texttt{b} - \texttt{c}$. The invariant can be verified by showing that each transition and any action allowed by the protocol maintains it. Furthermore, it is easy to show that the initial state satisfies this invariant; therefore, the invariant holds in all reachable states of the program.

Let *Spec* denote the specification for the merge element from equation 1, and let I denote the invariant from equation 7. It is straightforward to show that

$$I \quad \Rightarrow \quad \mathit{Spec} \tag{2}$$

Thus, the program of figure 15 correctly implements a merge element. There are many other useful consequences of I that guide our design. As noted above, $I \Rightarrow (\texttt{a} + \texttt{b} - \texttt{c}) \in [0, 1, 2]$. Thus, for any $M \geq 3$,

$$I \quad \Rightarrow \quad (\texttt{a} + \texttt{b} - \texttt{c}) = ((\texttt{a} + \texttt{b} - \texttt{c}) \bmod M) \tag{3}$$

This justifies the use of modulo-4 arithmetic in our implementation of the count-and-compare unit in figure 8.

We can also use I to show that the actions in our design are hazard-free. The specification of the merge-element is delay-insensitive; thus, the interfaces with the clients and the parent are hazard free. Furthermore,

$$\begin{array}{rcl} I & \Rightarrow & ((\texttt{a} \neq \texttt{\#req1}) \Rightarrow (\texttt{req1} \neq \texttt{ack1})) \\ & \wedge & ((\texttt{b} \neq \texttt{\#req2}) \Rightarrow (\texttt{req2} \neq \texttt{ack2})) \\ & \wedge & ((\texttt{c} \neq \texttt{\#ackP}) \Rightarrow (\texttt{ackP} \neq \texttt{reqP})) \end{array} \tag{4}$$

Thus, no action for incrementing a counter in the count-and-compare unit is disabled before it is performed.

The transitions that control the `eq` signal are a bit more complicated. Because $(\texttt{a} + \texttt{b} + \texttt{c}) \geq 0$, the only transition that can set `eq` to true is the one that increments `c`, and the only ones that can set `eq` to false are those that increment `a` and `b`. Noting that

$$\begin{array}{rl} & (I \wedge \neg\, \texttt{eq} \wedge (\texttt{reqP} = \texttt{xP})) \\ \Rightarrow & (\texttt{c} = \texttt{\#ackP}) \wedge (\texttt{xP} = \texttt{ackP}) \end{array} \tag{5}$$

we conclude that the latch that produces `reqP` is free from hazards. The same cannot be said for the latches that produce `ack1` and `ack2`. We can show

$$\begin{array}{rcl} I & \Rightarrow & ((\texttt{ack1} \neq \texttt{x1}) \Rightarrow (\texttt{x1} = \texttt{req1})) \\ & \wedge & ((\texttt{ack2} \neq \texttt{x2}) \Rightarrow (\texttt{x2} = \texttt{req2})) \end{array} \tag{6}$$

However, the `eq` signal is not guaranteed to remain high when these latches are enabled. Instead, we draw the weaker conclusion that the `x1` and `x2` inputs of these latches remain constant from any enabling of the latch until it finally performs its action. It is this observation that motivates the Schmitt-trigger based design shown in figure 6.

```
ASSUME: δ > δ_cc

INITIALLY:
     ¬(req1 ∨ req2 ∨ reqP ∨ ack1 ∨ ack2 ∨ ackP)
   ∧ ¬(x1 ∨ x2 ∨ xP) ∧ eq) ∧ (τ = 0)
   ∧ (req1.τ = −δ) ∧ (req2.τ = −δ)
   ∧ (ackP.τ = −δ)

PROTOCOL:
     (τ' ≥ τ)
   ∧ (a ≠ #req1) ⇒ (τ' ≤ req1.τ + δ_cc)
   ∧ (b ≠ #req2) ⇒ (τ' ≤ req2.τ + δ_cc)
   ∧ (c ≠ #ackP) ⇒ (τ' ≤ ackP.τ + δ_cc)
TRANSITIONS:
   (* environment actions *)
     ≪ req1 = ack1 → req1, req1.τ, #req1
          := NOT ack1, τ, #req1+1 ≫
   ∥ ≪ req2 = ack2 → req2, req2.τ, #req2
          := NOT ack2, τ, #req2+1 ≫
   ∥ ≪ ackP ≠ reqP → ackP, ackP.τ, #ackP
          := reqP, τ, #ackP+1 ≫

   (* count and compare actions *)
   ∥ ≪ a ≠ #req1 → a, eq :=
          a+1, a+b+1 = c ≫
   ∥ ≪ b ≠ #req2 → b, eq :=
          b+1, a+b+1 = c ≫
   ∥ ≪ c ≠ #ackP → c, eq :=
          c+1, a+b-1 = c ≫

   (* delay line actions *)
   ∥ ≪ (τ > req1.τ + δ_2) AND (x1 ≠ req1)
          → x1 := req1 ≫
   ∥ ≪ (τ > req2.τ + δ_2) AND (x2 ≠ req2)
          → x2 := req2 ≫
   ∥ ≪ (τ > ackP.τ + δ_1) AND (xP ≠ ackP)
          → xP := ackP ≫

   (* generate ack1, ack2, and reqP events *)
   ∥ ≪ eq AND (ack1 ≠ x1) →
          ack1, #ack1 := x1, #ack1+1 ≫
   ∥ ≪ eq AND (ack2 ≠ x2) →
          ack2, #ack2 := x2, #ack2+1 ≫
   ∥ ≪ (NOT eq) AND (reqP = xP) →
          reqP, #reqP := NOT xP, #reqP+1 ≫
```

Figure 14. Synchronized Transitions Program for the Schmitt-Trigger Based Merge

$$
\begin{aligned}
&(\;(\quad (\mathrm{req1} = \mathrm{ack1}) \wedge (\mathrm{x1} = \mathrm{req1}) \\
&\qquad \wedge (\#\mathrm{req1} = \#\mathrm{ack1}) \wedge (\tau \geq \mathrm{req1}.\tau + \delta)) && (7.1)\\
&\quad \vee (\quad (\mathrm{req1} \neq \mathrm{ack1}) \wedge (\mathrm{x1} \neq \mathrm{req1}) \\
&\qquad \wedge (\#\mathrm{req1} = \#\mathrm{ack1} + 1) \wedge (\tau \geq \mathrm{req1}.\tau)) && (7.2)\\
&\quad \vee (\quad (\mathrm{req1} \neq \mathrm{ack1}) \wedge (\mathrm{x1} = \mathrm{req1}) \\
&\qquad \wedge (\#\mathrm{req1} = \#\mathrm{ack1} + 1) \\
&\qquad \wedge (\tau \geq \mathrm{req1}.\tau + \delta))) && (7.3)\\
\wedge &(\;(\quad (\mathrm{req2} = \mathrm{ack2}) \wedge (\mathrm{x2} = \mathrm{req2}) \\
&\qquad \wedge (\#\mathrm{req2} = \#\mathrm{ack2}) \wedge (\tau \geq \mathrm{req2}.\tau + \delta)) && (7.4)\\
&\quad \vee (\quad (\mathrm{req2} \neq \mathrm{ack2}) \wedge (\mathrm{x2} \neq \mathrm{req2}) \\
&\qquad \wedge (\#\mathrm{req2} = \#\mathrm{ack2} + 1) \wedge (\tau \geq \mathrm{req2}.\tau)) && (7.5)\\
&\quad \vee (\quad (\mathrm{req2} \neq \mathrm{ack2}) \wedge (\mathrm{x2} = \mathrm{req2}) \\
&\qquad \wedge (\#\mathrm{req2} = \#\mathrm{ack2} + 1) \\
&\qquad \wedge (\tau \geq \mathrm{req2}.\tau + \delta))) && (7.6)\\
\wedge &(\;(\quad (\mathrm{reqP} = \mathrm{ackP}) \wedge (\mathrm{xP} = \mathrm{ackP}) \\
&\qquad \wedge (\#\mathrm{reqP} = \#\mathrm{ackP}) \wedge (\tau \geq \mathrm{ackP}.\tau + \delta)) && (7.7)\\
&\quad \vee (\quad (\mathrm{reqP} \neq \mathrm{ackP}) \wedge (\mathrm{xP} = \mathrm{ackP}) \\
&\qquad \wedge (\#\mathrm{reqP} = \#\mathrm{ackP} + 1) \\
&\qquad \wedge (\tau \geq \mathrm{ackP}.\tau + \delta)) && (7.8)\\
&\quad \vee (\quad (\mathrm{reqP} = \mathrm{ackP}) \wedge (\mathrm{xP} \neq \mathrm{ackP}) \\
&\qquad \wedge (\#\mathrm{reqP} = \#\mathrm{ackP}) \wedge \neg\,\mathrm{eq} \wedge (\tau \geq \mathrm{req2}.\tau))) && (7.9)\\
\wedge &(\quad (\mathrm{a} = \#\mathrm{req1}) \\
&\quad \vee ((\mathrm{a} = \#\mathrm{req1} - 1) \wedge (\tau \leq \mathrm{req1}.\tau + \delta_{\mathrm{CC}}))) && (7.10)\\
\wedge &(\quad (\mathrm{b} = \#\mathrm{req2}) \\
&\quad \vee ((\mathrm{b} = \#\mathrm{req2} - 1) \wedge (\tau \leq \mathrm{req2}.\tau + \delta_{\mathrm{CC}}))) && (7.11)\\
\wedge &(\quad (\mathrm{c} = \#\mathrm{ackP}) \\
&\quad \vee (\quad (\mathrm{c} = \#\mathrm{ackP} - 1) \wedge \neg\,\mathrm{eq} \\
&\qquad \wedge (\tau \leq \mathrm{ackP}.\tau + \delta_{\mathrm{CC}}))) && (7.12)\\
\wedge &(\;(\quad (\mathrm{a} + \mathrm{b} = \mathrm{c}) \wedge \mathrm{eq} \\
&\qquad \wedge (\mathrm{c} = \#\mathrm{ackP}) \wedge (\mathrm{reqP} = \mathrm{ackP})) && (7.13)\\
&\quad \vee (\quad (\mathrm{a} + \mathrm{b} = \mathrm{c} + 1) \wedge \neg\,\mathrm{eq} \\
&\qquad \wedge (\quad ((\mathrm{a} = \#\mathrm{req1}) \wedge (\mathrm{ack1} \neq \mathrm{req1})) \\
&\qquad\quad \vee ((\mathrm{b} = \#\mathrm{req2}) \wedge (\mathrm{ack2} \neq \mathrm{req2})))) && (7.14)\\
&\quad \vee (\quad (\mathrm{a} + \mathrm{b} = \mathrm{c} + 2) \wedge \neg\,\mathrm{eq} \\
&\qquad \wedge (\mathrm{a} = \#\mathrm{req1}) \wedge (\mathrm{b} = \#\mathrm{req2}))) && (7.15)
\end{aligned}
$$

Figure 15. Invariant for the Schmitt-Trigger Based Merge