A Pausible Bisynchronous FIFO for GALS Systems

Ben Keller*†, Matthew Fojtik*, and Brucek Khailany*
*NVIDIA Corporation †University of California, Berkeley
bkeller@eecs.berkeley.edu, mfojtik@nvidia.com, bkhailany@nvidia.com

***Abstract*—Many of the challenges of modern SoC design can be mitigated or eliminated with globally asynchronous, locally synchronous (GALS) design techniques. Partitioning a design into many synchronous islands introduces myriad asynchronous boundary crossings which typically incur high latency. We have designed a pausible bisynchronous FIFO that achieves low interface latency with a pausible clocking scheme. While traditional synchronizers have a non-zero probability of metastability and error, pausible clocking enables error-free operation by permitting infrequent slowdowns in the clock rate. Unlike prior pausible synchronizers, our circuit employs standard two-ported synchronous FIFOs, common circuit elements that integrate well with standard toolflows. The pausible bisynchronous FIFO achieves an average latency of 1.34 cycles across an asynchronous interface while using less energy and area than traditional synchronizers.**

I. Introduction

Modern SoCs built in deeply scaled process nodes present extraordinary design challenges. Slow wires and process, voltage, and temperature (PVT) variation make the synchronous abstraction increasingly untenable over large chip areas, requiring immense effort to achieve timing closure. The globally asynchronous, locally synchronous (GALS) design methodology is one means of mitigating the difficulty of global timing closure. GALS design flows delimit ”synchronous islands” of logic that operate on local clocks and communicate with each other asynchronously.

GALS has a decades-long history in academia [1], and the use of multiple clock domains is common in industry today [2] [3]. However, individual clock domains in large commercial designs still span many square millimeters, and so many of the design challenges posed by a fully synchronous design persist in these systems. The full advantages of GALS design can only be realized if large SoCs are partitioned into myriad small synchronous blocks, not a handful of large areas, an approach we refer to as *fine-grained* GALS. Industry has been reluctant to adopt this approach due to three main issues: the difficulty of generating many local clocks, the latency incurred by asynchronous boundary crossings, and the challenge of integrating GALS methodology into standard toolflows. Overcoming these difficulties will permit widespread adoption of GALS and ease the timing closure challenge in large SoCs.

We propose a novel design for low-latency asynchronous boundary crossings using pausible clocks. Our interface uses the standard two-port first-in first-out (FIFO) queues common in digital designs, but synchronizes read and write pointer updates using two-phase signals that allow data to traverse the interface with very low latency. The contributions of this paper include:

1) A novel flow-control scheme for FIFO pointers that uses two-phase increment and acknowledge signals to transmit data across an asynchronous interface.
2) A low-latency bisynchronous FIFO design that uses pausible clocking techniques with standard two-ported synchronous FIFOs that integrate easily into standard toolflows.
3) A thorough analysis of the timing constraints imposed by pausible clocking systems, including consideration of the delay required for signals to traverse the distance between the interface and the clock generator circuit.

We believe that this work overcomes many of the barriers to the adoption of fine-grained GALS in modern SoCs.

This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA). The views, opinions, and/or findings contained in this article are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement A (approved for public release, distribution unlimited).

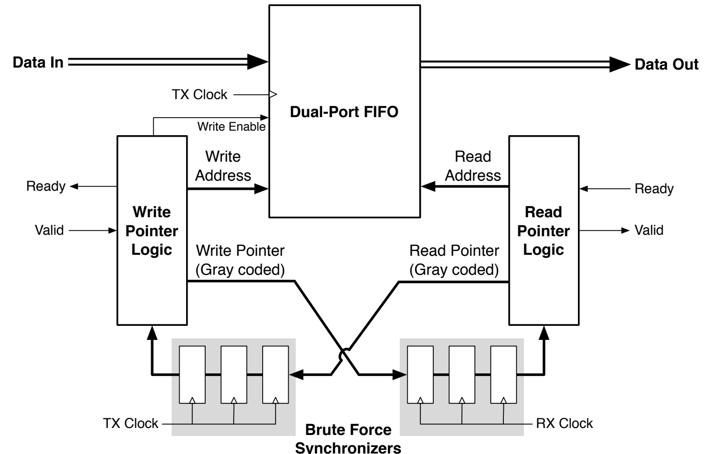

Fig. 1. A brute-force bisynchronous FIFO.

II. Background

While industry has not yet embraced the GALS approach, progress has been made in overcoming the barriers to GALS. Previous work addresses some of the challenges of local clock generation and synchronization latency.

A. Local Clock Generation

Historically, on-chip clocks have typically been generated by phase-locked loop (PLL) circuits. These circuits can reliably generate a fixed target frequency, but are large, power-hungry, and difficult to design, making them poor candidates for inclusion in each synchronous island of a GALS system. Recently, some systems have abandoned the goal of a fixed target frequency in favor of adaptive clocking schemes that can temporarily vary the clock period in response to noise events [4]. Going further, some adaptive clocks do not target

a particular frequency at all, instead using replica critical path circuits to continuously adjust the generated clock as local conditions change [5]. Since they do not attempt to lock to particular frequency targets, adaptive clock generators avoid much of the complexity of PLL circuits. These circuits impose minimal overhead on the overall system, and are ideal candidates for local clock generation in GALS designs [6].

B. Synchronization Latency

Signals crossing the boundary between fully asynchronous clock domains, such as those crossing between synchronous islands in a GALS design, must be synchronized to minimize the risk of metastability and operational failure. This synchronization is typically achieved by sending such signals through several flip-flops in series in the receiver clock domain. The flip-flops delay the signal for one or more cycles, providing extra time for any metastability to resolve. While these *brute force* (BF) synchronizers do not eliminate the possibility of metastability, they can reduce the probability until it is negligible. BF synchronizers can be used with a FIFO memory to construct a *BF bisynchronous FIFO* as shown in Figure 1. This FIFO safely transmits data between two clock domains, synchronizing the read and write pointers with BF synchronizers. The pointers must be gray coded so that any synchronization error does not disrupt the pointer location by more than one increment; the logic to encode and decode the pointers is an overhead of this scheme.

Several circuits have been designed with the explicit purpose of reducing the synchronization latency penalty. Chakraborty and Greenstreet demonstrate circuits that can synchronize data with low latency if some information about the relative phase of the two clocks is known at design time [7]. However, their scheme is not practical in the case of fully asynchronous clock domains. Dally and Tell devised an even/odd synchronizer that achieves low-latency communication across an asynchronous interface, but their circuit requires a complicated phase predictor and functions only with stable clock frequencies [8]. These circuits are useful in certain applications, but do not provide a satisfactory solution to the barriers to GALS adoption.

C. Pausible Clocking

A different method to reduce synchronization latency is pausible clocking. As described in [9], pausible clocks take advantage of the adaptive clock circuits already present in many GALS implementations. A simple adaptive clock circuit consists of one or more inverting delay lines fed into the input of a Muller C-element (see Figure 2a). These delay lines replicate the various critical paths found in the synchronous logic island; the C-element ensures that the next clock edge will not be generated until the slowest replica path resolves. The pausible clock circuit adds another input to the C-element that can be triggered asynchronously by signals entering the clock domain (see Figure 2b). A mutual exclusion (mutex) circuit ensures that the asynchronous input cannot toggle simultaneously with the rising clock edge. When the mutex input r2 is high, the mutex is opaque and signals at the data input r1 are delayed until the clock edge has passed. When r2 is low, the mutex is transparent and signals at r1 are passed through. Because the mutex can become metastable if its inputs toggle simultaneously, the clock can pause for an arbitrarily long duration (with vanishingly small probability) if r1 and r2 go high at the same time. However, there is no longer any danger of metastability at the asynchronous input, and typical circuit operation synchronizes input signals with roughly one cycle of latency. Prior work describes the design and operation of pausible clocking circuits in detail [6] [9] [10].

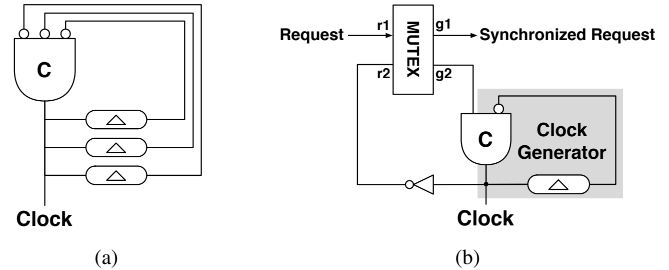

Fig. 2. Adaptive clock generators (a) can be extended with a mutex to synchronize requests, forming a pausible clock circuit (b).

D. Related Work

Pausible clocking enables low-latency synchronization of signals with arbitrary relative phase, and as such represents an attractive option for boundary crossings in GALS design. Several prior proposals for GALS boundary crossings integrate pausible clocks with FIFO queues to synchronize data across an interface [6] [9] [10] [11] [12]. These designs typically require a fully asynchronous FIFO that services two-phase request and acknowledge signals to store data words in transit. There are many asynchronous FIFO designs in the literature, from Sutherland's classic micropipelines [13] to GasP and Mousetrap FIFOs [14] [15]. However, these asynchronous FIFOs have several disadvantages over their synchronous counterparts. Rather than keeping data in place and updating pointers to the data, these FIFOs propagate data from the back to the front of the queue. This data movement incurs a penalty in both energy and latency, a penalty that increases with the queue depth. Furthermore, many asynchronous FIFOs require careful delay matching to satisfy two-sided timing constraints. Some of these issues can be mitigated by the use of circular FIFOs, such as the one proposed in [16]. However, asynchronous FIFOs necessarily require careful asynchronous circuit design and verification that is poorly supported by standard VLSI toolflows.

III. The Pausible Bisynchronous FIFO

We propose a pausible clocking scheme that achieves flow control via a standard two-ported synchronous memory element that is synchronously written in one clock domain and asynchronously read in another. We refer to this circuit as a *pausible bisynchronous FIFO*. Like the BF bisynchronous FIFO, data is stored in the FIFO while the read and write pointers are synchronized between clock domains. In the pausible FIFO, however, this synchronization is completed with a pausible clock network, not with slow BF synchronizers. This design combines the low-latency synchronization of pausible clocking with the favorable characteristics of standard two-ported FIFOs.

Figure 3 shows the pausible bisynchronous FIFO circuit. The pausible synchronizers that provide synchronization in both the transmit (TX) and receive (RX) clock domains are

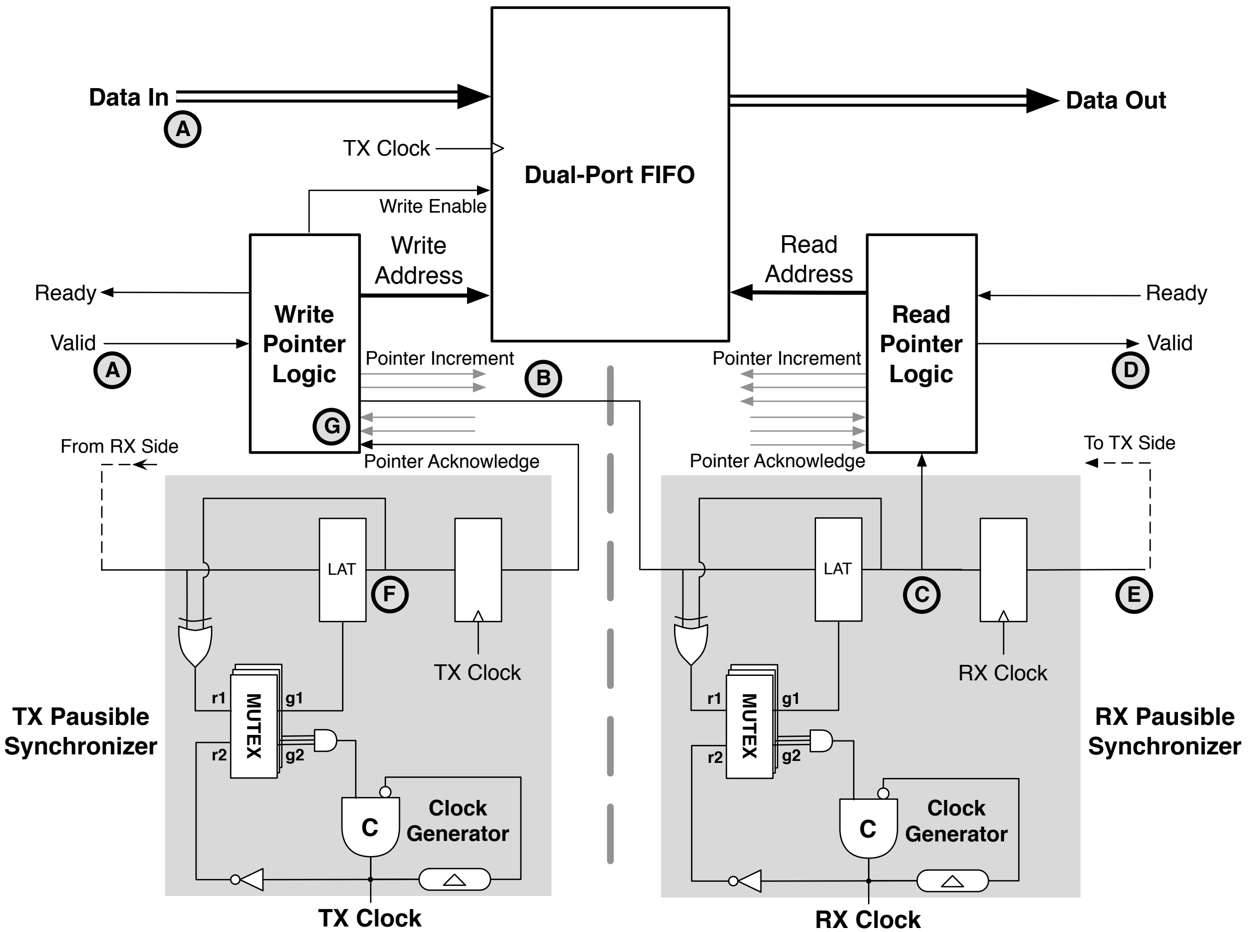

Fig. 3. The pausible bisynchronous FIFO. Only one of the increment-acknowledge paths is shown for clarity; in the complete system, each increment and acknowledge line requires its own mutex and synchronization circuitry. The labeled letters show the sequence necessary to synchronize data through the FIFO.

shown in gray. These circuits are similar to those in [6], except that the feedback FF has been replaced with a latch to reduce overhead. Note that each pausible clock circuit requires its input pointer increment or acknowledge signal to use a two-phase request-acknowledge protocol. This ensures that the unsynchronized signal can only toggle once, and then must wait for an acknowledgement before toggling again, preventing additional switching at an unsafe clock phase. By design, this protocol prevents multiple toggles within a single clock period; however, this is problematic for the synchronization of pointer updates, because it implies that each pointer can only be updated once per cycle, restricting throughput to the slower of the two clock periods. Accordingly, the pausible FIFO does not synchronize the multi-bit pointers directly. Instead, several single-bit, two-phase *pointer increment* lines signal an update to the read or write pointers, and corresponding *pointer acknowledge* signals are sent back once the increments are synchronized. This allows multiple pointer increments to occur in succession within a single clock period, and allows full throughput even at mismatched clock periods. Our experimentation found that three increment-acknowledge pairs in either direction guaranteed full throughput for TX:RX clock period ratios as high as 2 or as low as 1/2. Additional increment and acknowledge lines could be added to ensure full throughput in the case of more extreme mismatches between TX and RX clock periods, but this is likely unnecessary, as sending data across an interface with such mismatched periods would quickly fill or empty the FIFO.

Each of the increment and acknowledge signals must be synchronized through their own mutex circuit in the pausible clock network. The g2 outputs of all mutexes are ANDed together, and this result is used as the synchronizing input to the C-element, ensuring that the clock edge is not generated until every mutex guarantees a safe phase. Additional interfaces (e.g., to multiple different synchronous islands) can also be accommodated in this way: the g2 outputs from every interface can be ANDed together to ensure that all interfaces synchronize correctly. This does have the side effect that a clock pause from any one interface will stall the entire synchronous domain, but we found clock pauses to be so rare that we do not believe this will pose a significant problem in practice (see Section VI).

The write pointer logic stores the value of the write pointer, as well as its best knowledge of the read pointer (possibly delayed from the actual read pointer position as updates are synchronized from the RX domain). It uses these values to calculate whether the FIFO is full, and to signal backpressure accordingly. The write pointer logic also sends write pointer increment signals by toggling one of the two-phase write pointer increment lines in the event of a write to the FIFO.

TABLE I. TIMING VARIABLES

Variable	Description
T	The nominal clock period of the synchronous block.
T_L	The average latency of a data word through the interface.
t_{ins}	The insertion delay of the clock for the synchronous block.
t_{r2}	The delay from the output of the C-element to the mutex r2 input.
t_{fb}	The delay from the mutex r2 input through the mutex and around the feedback path to the mutex r1 input.
t_{g2}	The delay from the mutex r1 input through the mutex to the output of the C-element.
t_{CL}	The minimum time available to perform combinational work on the synchronized request signal before the next clock edge.
t_m	Time allotted to resolve mutex metastability, used to reduce the frequency of clock pauses.
t_w	The wire delay from the boundary of the synchronous island to the local clock generator.

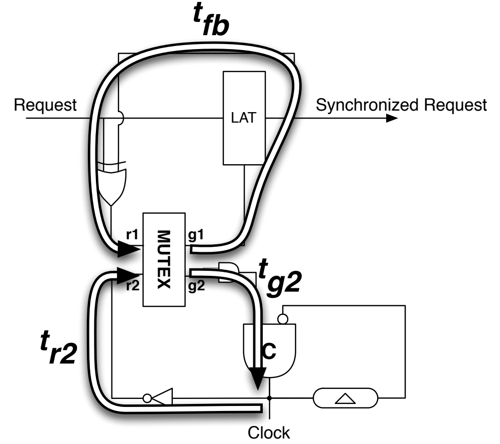

Fig. 4. The key timing paths in the pausible synchronizer.

A state machine tracks which increment signals are in flight and which have been acknowledged and can be used again. The read pointer logic performs similar calculations in the RX clock domain to determine whether the FIFO is empty. With this logic and the pausible clocks synchronizing the pointer updates, the pausible bisynchronous FIFO can synchronize new input data in roughly one cycle on average.

The dual-port FIFO is clocked by the TX clock, and can be implemented as FFs, a latch array, or an SRAM macro as appropriate for its size. Such FIFOs are standard circuit elements in modern designs, and the numerous area and energy optimizations for these memory elements can be leveraged with no additional design effort. No custom design is needed to implement the FIFO, and standard scan and test structures can be easily implemented.

The pausible bisynchronous FIFO could be easily modified to interface between a clock domain with pausible clocking and one with a traditional fixed reference, such as a PLL. By replacing the pausible synchronizer on the fixed-reference side of the interface with brute-force synchronizing FFs to synchronize the increment and acknowledge pointers, low latency in one direction would still be maintained. This would allow a system to be partially converted to a GALS style while maintaining legacy IP with traditional clocking where necessary. These advantages make the pausible bisynchronous FIFO a good candidate to overcome the barriers to widespread GALS adoption.

IV. CIRCUIT OPERATION

The sequence labeled on Figure 3 shows the series of steps involved in writing a data word to the FIFO. In this example, the FIFO is initially empty, and all two-phase increment and acknowledge lines are available for use. On the rising edge of the TX clock, data is written to the FIFO address pointed to by the write pointer and the input valid signal is asserted (A). At this point, data is available to be read out of the FIFO. The write pointer logic then increments the write pointer internally, and toggles one of the two-phase write pointer increment lines (B). This write pointer increment line is toggled in the TX domain, and so it is asynchronous to the RX domain and must be synchronized through the RX pausible clock network. Depending upon the phase at which the write pointer increment toggle arrives at the RX domain, it may pass through immediately, be delayed until after the next RX clock edge, or (in rare cases) cause metastability in the mutex and be delayed for a longer time. However, the update will eventually be synchronized into the RX domain (C), at which point the read pointer logic can increment its internal tracking of the write pointer and assert the valid signal at the output of the system (D).

At this point, the data can be synchronously read from the FIFO in the RX domain. (Once this read occurs, the RX pointer logic will need to toggle one of the read pointer increment signals to inform the TX domain; this series of toggles is not shown.) However, from the perspective of the TX domain, the write pointer update is still in flight, as a corresponding acknowledge signal has not yet been received. Accordingly, after the synchronization, the RX clock edge toggles the corresponding acknowledge line (E). As this toggle occurs in the RX clock domain, it must be synchronized through the TX pausible clock network (F). This acknowledge signal then updates the TX logic state machine, freeing the write pointer increment line for future use (G).

V. TIMING ANALYSIS

Pausible clocking integrates the logic for asynchronous boundary crossings into the clock generation mechanism for the entire synchronous island. This integration imposes constraints on the operating conditions of each of these systems. Previous work in pausible clocks does not fully address these constraints, but this paper contributes a thorough accounting of the capabilities and limitations of pausible clock timing, which is critical to designing a realistic system. In this section, we derive expressions for the average latency of the pausible interface, as well as the constraints imposed upon the clock period, insertion delay, and wire delay across the synchronous island. We neglect the effects of variation in this analysis, treating circuit delays as fixed quantities. In reality, stochastic or worst-case corner analysis would be needed to ensure timing robustness, although post-silicon tuning could alleviate the effects of process variation. Table I describes each of the variables used in the analysis in this section.

A. Timing Fundamentals

The important delays through the pausible clock network are shown in Figure 4. t_{r2} is the delay from the output of the C-element to the mutex r2 input. t_{fb} is the delay from the r2

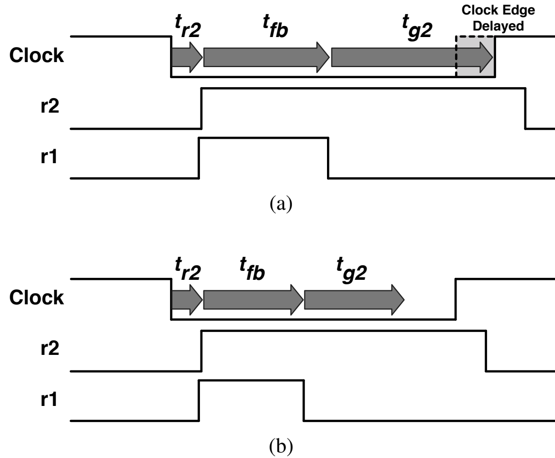

Fig. 5. The clock period timing constraint of pausible clocking. In the worst case, a request arrives just before r2 goes high. In (a), the sum of the delays through the pausible circuit is longer than half the clock period, and so the next clock edge is delayed. In (b), the delays are shorter and the clock edge occurs on time.

input through the mutex and around the feedback loop to the r1 input. t_{g2} is the delay from the mutex r1 input to the output of the C-element, including delay through the AND tree when multiple mutexes contribute timing information. The sum of these three delays cannot exceed the delay through the clock generator, or else the clock will frequently pause, increasing the clock period beyond the target for the synchronous island (see Figure 5). Since the clock generator delay is set to $T/2$ for a desired clock period T, these delays collectively enforce a minimum clock period for the synchronous block:

$$T/2 \geq t_{r2} + t_{fb} + t_{g2} \tag{1}$$

If this clock period constraint is exceeded, then the timing slack in the system translates into a margin t_m that guards against the effect of clock pauses:

$$t_m = T/2 - (t_{r2} + t_{fb} + t_{g2}) \tag{2}$$

Mutex metastability can be seen as a temporary increase in t_{fb} caused by simultaneous toggling of the mutex inputs. If (1) is just satisfied (that is, if $T/2 = t_{r2} + t_{fb} + t_{g2}$), then $t_m = 0$, and any mutex metastability that delays its output will cause the clock to pause. If $t_m > 0$, then some metastability can be tolerated before a clock pause occurs. In practice, we found mutex metastability to be an infrequent event, with long clock pauses rare (see Section VI). Accordingly, in this analysis we will tend to trade off t_m in favor of other more critical timing parameters.

As detailed in Section IV, the low latency of the pausible bisynchronous FIFO depends on the ability of the RX pointer logic to immediately respond to a write pointer update by asserting data valid before the next RX clock edge arrives. The worst-case setup time for this logic is shown in Figure 6. We refer to this available time to complete combinational work within the same cycle as a received request as t_{CL}. In the worst case for this timing path, metastability in the mutex causes a clock pause before resolving in favor of r1. When g1 toggles, a clock edge will be generated as soon as this signal propagates around the feedback loop to the clock generator. Thus, the time

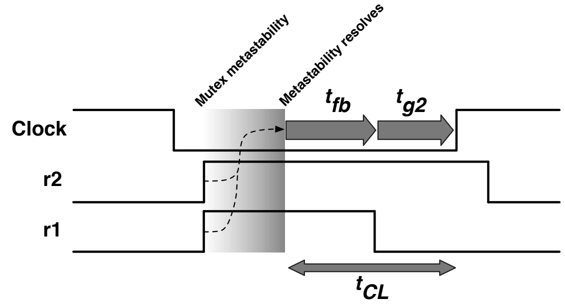

Fig. 6. The setup time requirement for the synchronized request signal. In the worst case, metastability consumes any available timing margin, so that when g1 finally goes high, the next clock edge is guaranteed to occur as soon as the signal can propagate through t_{fb} and t_{g2}. This is therefore the maximum allowable delay for logic that depends on the synchronized output.

T_{CL} available for logic before this clock edge is only

$$t_{CL} = t_{fb} + t_{g2}. \tag{3}$$

This parameter is constrained by the complexity of the pointer logic; if a long enough time is not apportioned for t_{CL}, then an extra register must be inserted before the logic to "pipeline" the computation, increasing the latency of the interface by one cycle. If $t_m > 0$, then increasing t_{fb} by adding delay to the feedback path trades off excess t_m to increase the time available for same-cycle combinational work.

In order to derive the average latency of the interface, the phase at which the request signal arrives must be considered. As shown in Figure 7, if a request signal arrives while the mutex is transparent, the request can be serviced within the same cycle. Assuming that the fully asynchronous request signal is equally likely to arrive at any phase, the average latency of such requests is $0.75T - t_{r2}$. If the request arrives while the mutex is opaque, then the request cannot be serviced until the next cycle. The average latency of such requests is $1.25T - t_{r2}$. If the duty cycle of the clock is 50%, then taking the mean of these two expressions gives the average latency t_L of the interface as a whole:

$$t_L = T - t_{r2} \tag{4}$$

Increasing t_{r2} decreases the average latency of the interface because it shifts the transparent phase of the mutex closer to the next clock edge. If $t_m > 0$, then increasing t_{r2} by adding delay to the mutex r2 input trades off excess t_m to decrease the average latency through the interface. Since t_m can also be traded for additional t_{CL}, this means that there is a trade-off between reducing latency and increasing the time available for combinational work in the read pointer logic.

B. Insertion Delay

In real systems, the clock distribution network within the synchronous island will have some insertion delay t_{ins} between the generation of the clock edges and their propagation through the clock network to the register endpoints. As first noted in [17], this insertion delay mis-aligns the mutex transparent phase, which could lead to circuit failure (see Figure 8). Small insertion delays can be compensated by intentionally increasing t_{r2} to match t_{ins}, realigning the phases and protecting against metastability. However, the clock period

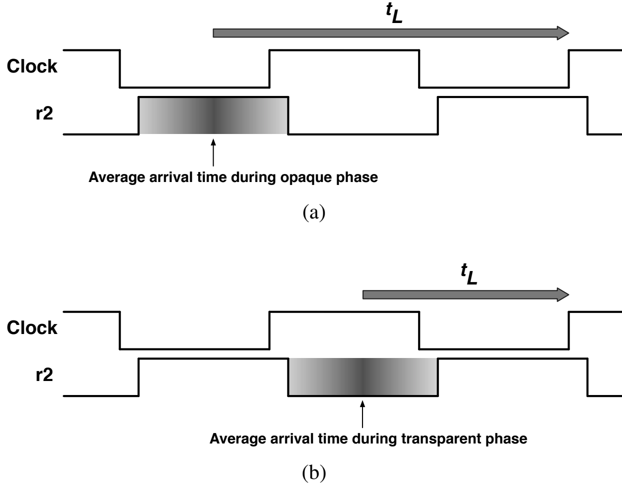

Fig. 7. The latency of the interface. Possible arrival times for each case are shaded. Data is generally available to be read out of the FIFO at the positive clock edge after the request passes through the synchronizer. Data that arrives during the opaque phase of the mutex (a) will average more than one cycle of latency. Data that arrives during the transparent phase of the mutex (b) will average less than one cycle of latency.

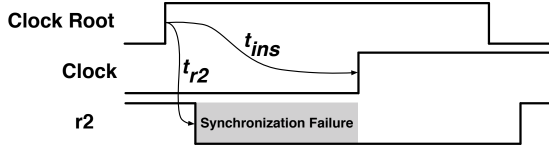

Fig. 8. The effect of insertion delay on pausible clock timing. If t_{r2} is small relative to the insertion delay, then the mutex is transparent when the clock edges arrive at the FFs. If requests arrive during the shaded period, they will pass through the mutex and may induce metastability in the FF following the pausible synchronizer.

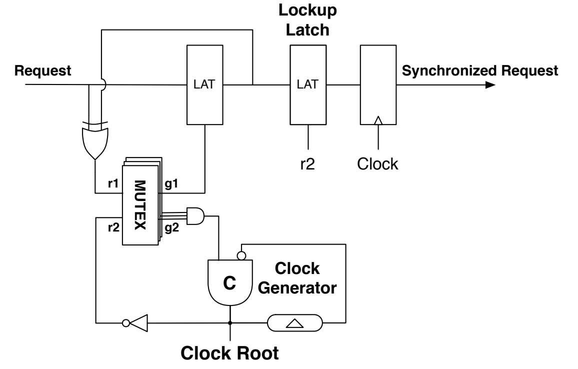

Fig. 9. The pausible synchronizer with an added lockup latch to guard against races caused by large insertion delays. The latch guards the clocked FF until after the clock can propagate from the root through the clock tree.

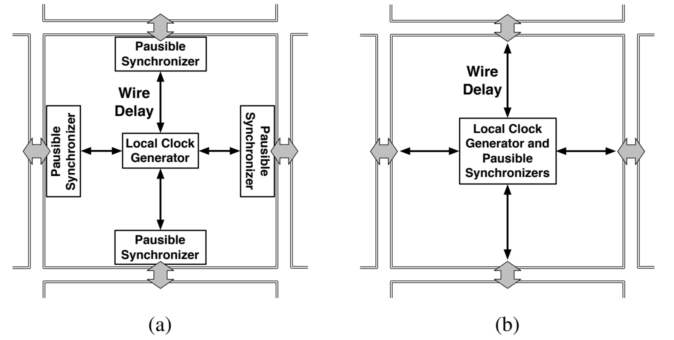

Fig. 10. The delays imposed on the pausible clocking system by the wire delay incurred to traverse each synchronous island. The synchronizer circuits can either be placed near the boundary (a) or near to the local clock generator (b).

constraint from (1) limits the increase in t_{r2}. Setting $t_{r2} = t_{ins}$ yields the constraint on the insertion delay permitted for a given clock period:

$$t_{ins} \leq T/2 - t_{fb} - t_{g2} \tag{5}$$

Handling large insertion delays is a fundamental challenge of pausible clocking schemes. One technique to allow larger insertion delay places all FFs adjacent to the interface on a separate clock with a much smaller clock tree [6]. However, this approach could pose challenges with standard toolflows. [18] proposed adding lockup latches to the circuit as in Figure 9. We propose a similar scheme, except that our transparent high latches are enabled by the r2 input, so they are transparent only when the mutex is not. The latches allow requests to propagate through the transparent mutex before the clock signal arrives at the leaf nodes, but then delays the request at the transparent mutex until after the clock edge has safely arrived at the FF clock input. The latches do not increase the latency of the interface because signals that would not race the clock would still have to wait for the next clock edge to be synchronized. Adding latches marginally increases the area and energy of the circuit, but allows an additional $T/2$ of insertion delay:

$$t_{ins} \leq T - t_{fb} - t_{g2} \tag{6}$$

However, t_{CL} is decreased by the delay through the transparent latch, as the asynchronous request must propagate through the latch before it reaches the combinational network.

Expressions for average latency and t_{CL} must be adjusted when insertion delay is considered:

$$t_L = T + t_{ins} - t_{r2} \tag{7}$$

$$t_{CL} = T/2 + t_{ins} - t_{r2} \tag{8}$$

Increasing insertion delay increases the average latency of the interface because the next clock edge is delayed relative to the transparent phase of the mutex. This delay also increases the time available for combinational work.

C. Wire Delay

In addition to the non-idealities of clock insertion delay, a nonzero wire delay is required to traverse the physical distance between the block interface and the clock generation circuit. Prior analysis of pausible clock timing neglects this delay, but collocating all of the blocks involved is impractical for a real GALS system. For instance, a tiled partitioning with a nearest-neighbor communication scheme would require four interfaces for each block, each communicating to a different neighbor (see Figure 11). It would not be physically possible to collocate the clock generator with these different interfaces.

Different design decisions impose this wire delay on different paths in the pausible clock circuit. A traditional approach places each mutex at the boundary of the synchronous

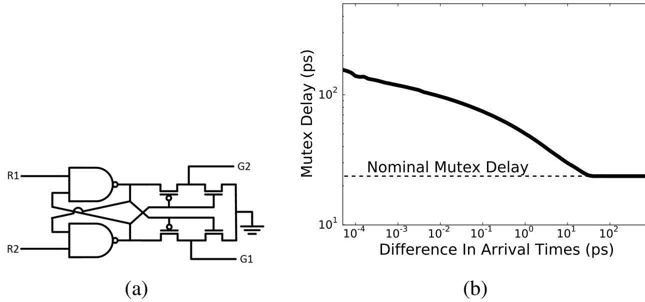

Fig. 11. A mutual exclusion (mutex) circuit. The circuit was simulated in SPICE to find the magnitude of mutex metastability as a function of the arrival time difference between signals r1 and r2.

island, with the local clock generator centrally located as in Figure 10a. This adds a wire delay t_w to t_{r2} and t_{g2}, increasing the minimum achievable cycle time (from (1)) and decreasing the maximum allowable insertion delay (from (5)). Alternately, all mutexes can be placed near the clock generator as in Figure 10b. This adds t_w to the latency of the system, but does not impact the cycle time or insertion delay constraints. In either approach, t_w could be reduced by using higher metal layers and dedicated routing channels to transmit these critical signals. Even with these considerations, t_w will likely be a substantial fraction of the clock period for most systems, and will therefore have a noticeable impact on system performance.

VI. Impact of Mutex Metastability

The above analysis assumed that mutex metastability is so rare that its impact on average latency and cycle time will be negligible. To confirm this assumption, we simulated the mutex circuit in Figure 11a in SPICE in a 28nm CMOS process. Inputs were toggled with random relative arrival times and the resulting delay through the circuit was measured. Figure 11b shows the results of the simulation. Assuming that the relative arrival times are uniformly distributed, integrating under the curve reveals that a 1ns clock period would be increased by an average of just 0.23ps from clock pauses, an impact of less than 0.1%. Furthermore, long clock pauses are exceedingly rare: pauses longer than 100ps make up less than one event in 10^6. Adding a small t_m eliminates most clock pauses and reduces the average impact on cycle time even further.

VII. Experimental Setup

To evaluate the pausible bisynchronous FIFO, we implemented the circuit as an interface module in Verilog RTL. Simple transmit and receive blocks were constructed to send and receive data; the interface manages communication between these two blocks (see Figure 12). The communication between the interface and its neighbors is fully synchronous, and uses standard ready-valid interfaces; all asynchronous communication takes place within the interface itself, similar to the "GALS wrapper" approach described in [18]. This allows other interfaces to be swapped in for the pausible bisynchronous FIFO for straightforward comparison. The mutex circuit and the local clock generators are described with behavioral Verilog, but the rest of the circuit is fully synthesizable.

To ensure that our circuit integrates well with standardized toolflows, we synthesized the circuit using Synopsys Design

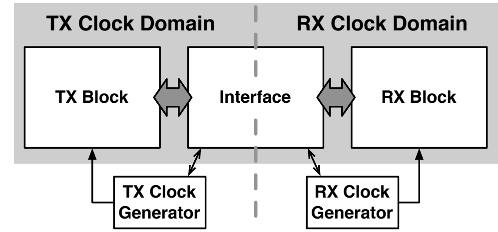

Fig. 12. The setup used to simulate the interfaces. The shaded area was synthesized to obtain area and energy comparisons.

Compiler in the same 28nm process as described in Section VI. The local clock generators were implemented as behavioral cells because the design of an adaptive clocking macro is outside the scope of this work. The mutex circuits were treated as black boxes with custom library definitions based roughly on SPICE simulations. They were defined as state elements for the synthesis tool so as to break the combinational loops inherent to pausible clock design. Unlike synthesis of a standard digital block, the pausible bisynchronous FIFO circuit has several timing paths that cross between clock domains. These paths should not be unconstrained; as noted previously, the timing of many of these paths is critical for circuit operation. Accordingly, we used the timing analysis from Section V to define custom timing constraints on these paths so the synthesis tool would appropriately optimize timing. We constrained the t_{fb} and t_{g2} paths to 200ps each, with t_{r2} negligible. According to (1), these constraints should permit operation with a clock period of 800ps or larger. An insertion delay of 250ps was explicitly added to the clock generation circuits, well below the maximum insertion delay permitted by (6). We did not insert additional wire delay for physical distance as described in Section V-C.

The post-synthesis netlist was simulated using VCS with delays annotated from synthesis. The probability of clock pausing was modeled with a simple exponential distribution estimated from the SPICE simulations described in Section VI, although the average impact of these clock pauses on latency was negligible. Power was measured by Primetime PX, with activity factors back-annotated from the gate-level simulation.

VIII. Performance Results

We compared the performance of three different interfaces:

- **Synchronous**, a fully synchronous FIFO queue that functions only within a single clock domain. This is a standard element in all digital designs. Since the read and write pointers do not have to be synchronized across clock domains, the control logic is simpler than either of the bisynchronous interfaces.
- **BFSync**, a brute force bisynchronous FIFO with three series FFs used to synchronize the pointers, as shown in Figure 1.
- **Pausible**, the pausible bisynchronous FIFO described in Section III and shown in Figure 3.

Each interface was used in the experimental setup shown in Figure 12. 128-bit data words were sent across the interface. Each interface included an 8-element FIFO built from FFs.

Figure 13 shows the average latency through the interface as the ratio of clock periods is varied. (The RX clock period is

TABLE II. SYNTHESIS RESULTS

	Average Latency (cycles)	Area (μm^2)	Power (mW)	Energy (fJ/bit)
Synchronous	1	4968	4.08	39.8
BFSync	4	5005	6.03	58.9
Pausible	1.34	4808	5.41	52.8

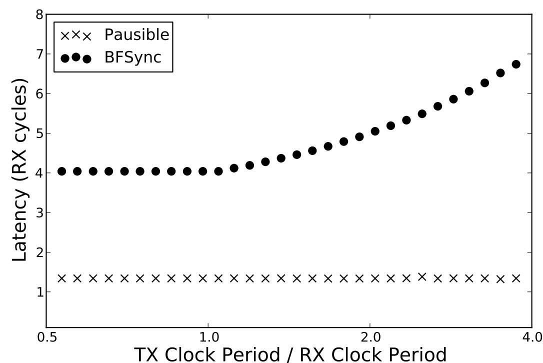

Fig. 13. Simulation results showing the latency of each interface as the ratio of TX clock period to RX clock period is varied. The latency of the pausible bisynchronous FIFO averages 1.34 cycles, and is much less than the latency through the brute-force synchronizer.

held fixed at 1.25ns, while the TX clock period is swept from 0.625ns to 5ns.) The pausible bisynchronous FIFO achieves an average latency of just 1.34 cycles throughout this range, much lower than the 4 cycles of BFSync and comparable to the 1-cycle synchronous latency. In accordance with (7), much of the increase in latency beyond one cycle is caused by 250ps insertion delay. The result is slightly higher than predicted by (7) because of the non-zero delay between the TX and RX interface estimated by the synthesis tool. If we include a wire delay t_w of 150ps in the simulation as described in Section V-C and assume the floorplan of Figure 10b, this delay is directly added to the latency of the interface. For a 1ns clock period, this increases the average latency from 1.34 cycles to 1.49 cycles.

Table II shows the results of synthesis of each design. The area of each design is dominated by that of the FIFO, so the total area of each design is similar. The energy cost of the pausible interface per bit of data sent is somewhat less than the BF synchronizer because the gray-coding logic is removed. The synthesized design is able to operate with a clock period as low as 800ps (as predicted by (1)) before clock pauses start to become much more frequent. The interface still operates correctly at a clock period of 600ps, although faster periods cause the setup time constraint in (8) to be violated, leading to incorrect functionality.

IX. CONCLUSION

We have designed a low-latency asynchronous interface that works well with standard design tools. The pausible bisynchronous FIFO achieves an average of 1.34 cycles of latency, while incurring minimal energy and area overhead over a synchronous interface. Careful analysis of the timing constraints imposed by the system allows full integration with standard toolflows. We believe that this circuit represents a key enabling technology for fine-grained GALS systems, which can mitigate many of the challenges of modern SoC design.

REFERENCES

[1] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," *Ph. D. Thesis*, vol. 1, p. 50, 1984.

[2] E. Fluhr *et al.*, "Power8: A 12-core server-class processor in 22nm soi with 7.6tb/s off-chip bandwidth," in *Proc. IEEE International Solid-State Circuits Conference*, 2014, pp. 96–97.

[3] S. Rusu *et al.*, "Ivytown: A 22nm 15-core enterprise xeon processor family," in *Proc. IEEE International Solid-State Circuits Conference*, 2014, pp. 102–103.

[4] A. Grenat *et al.*, "Adaptive clocking system for improved power efficiency in a 28nm x86-64 microprocessor," in *Proc. IEEE International Solid-State Circuits Conference*, 2014, pp. 106–107.

[5] R. Jevtic *et al.*, "Per-Core DVFS With Switched-Capacitor Converters for Energy Efficiency in Manycore Processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pp. 1–8, 2014.

[6] R. Mullins and S. Moore, "Demystifying Data-Driven and Pausible Clocking Schemes," in *Proc. IEEE Symposium on Asynchronous Circuits and Systems*, 2007, pp. 175–185.

[7] A. Chakraborty and M. Greenstreet, "Efficient Self-Timed Interfaces for Crossing Clock Domains," in *Proc. IEEE Symposium on Asynchronous Circuits and Systems*, 2003, pp. 78–88.

[8] W. J. Dally and S. G. Tell, "The Even/Odd Synchronizer: A Fast, All-Digital, Periodic Synchronizer," in *Proc. IEEE Symposium on Asynchronous Circuits and Systems*, 2010, pp. 75–84.

[9] K. Yun and R. Donohue, "Pausible clocking: a first step toward heterogeneous systems," in *Proc. IEEE International Conference on Computer Design*, 1996, pp. 118–123.

[10] P. Teehan *et al.*, "A Survey and Taxonomy of GALS Design Styles," in *Proc. IEEE Design & Test of Computers*, vol. 24, no. 5, 2007, pp. 418–428.

[11] S. Moore *et al.*, "Point to point GALS interconnect," in *Proc. IEEE Symposium on Asynchronous Circuits and Systems*, 2002, pp. 69–75.

[12] E. Tuncer *et al.*, "Enabling adaptability through elastic clocks," in *Proc. ACM/IEEE Design Automation Conference*, 2009, pp. 8–10.

[13] I. E. Sutherland, "Micropipelines," *Communications of the ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[14] I. Sutherland and S. Fairbanks, "GasP: a minimal FIFO control," in *Proc. IEEE Symposium on Asynchronous Circuits and Systems*, 2001, pp. 46–53.

[15] M. Singh and S. Nowick, "MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 684–698, Jun. 2007.

[16] A. Ghiribaldi *et al.*, "A Transition-Signaling Bundled Data NoC Switch Architecture for Cost-effective GALS Multicore Systems," *Proc. IEEE Design, Automation & Test in Europe*, pp. 332–337, 2013.

[17] A. E. Sjogren and C. J. Myers, "Interfacing synchronous and asynchronous modules within a high-speed pipeline," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 8, pp. 573–583, 2000.

[18] X. Fan *et al.*, "Analysis and optimization of pausible clocking based GALS design," *IEEE International Conference Computer Design*, 2009.