Synthesizable Synchronization FIFOs Utilizing the Asynchronous Pulse-Based Handshake Protocol

Ameer M.S. Abdelhadi
Department of Electrical and Computer Engineering
University of Toronto
Toronto ON M5S 3G4, Canada
ameer.abdelhadi@utoronto.ca

***Abstract*—We present a family of synthesizable standard-cell-based synchronization FIFOs for crossing between asynchronous and synchronous timing domains. These FIFOs are composed of modular mix-and-match components. The input and output interfaces are interchangeable for edge-triggered synchronous communication and for the asP* asynchronous pulse-based handshake protocol. The FIFO capacity, data width, synchronizer latency, and interface protocols are independent design parameters, allowing the FIFO to be easily configured for different requirements. The modular design makes the FIFO ideal for use in system-on-chip applications, where clock-domain crossing is required, or multiple clock domains communicate using an asynchronous network-on-chip. Our designs demonstrate high-throughput communication and hide most of the synchronization latency. The FIFOs are fully synthesizable using widely available standard-cell libraries and a standard ASIC design flow. We generate layouts for several different synchronous and asynchronous FIFOs. All instances can operate at speeds greater than 1.92 GHz under worst-case conditions when implemented in a 45nm CMOS process. The design flow and a fully parameterized Verilog implementation are released as an open-source library.**

***Index Terms*—networks-on-chip (NoC), synchronizing FIFO, synchronization, clock-domain crossing (CDC), globally asynchronous locally synchronous (GALS), asP* handshake protocol**

I. Introduction

Modern chip designs can consist of several billion transistors. Because of the difficulties of distributing high-speed clocks with low skew and jitter, such chips are invariably organized as hundreds of relatively independent timing domains. This approach leverages the mature, commercially supported, design flows for building synchronous modules with millions of gates, while providing a timing independence between these modules. This simplifies timing closure, supports design reuse, and enables independent voltage-frequency scaling to be used in separate modules to maximize energy efficiency. Globally, large chips are asynchronous. Therefore, optimizing the asynchronous interfaces between timing domains is essential for achieving efficient, high-performance systems. Current Systems-on-chip designs are partitioned into multiple clock domains and can involve large numbers of clock-domain crossings. This motivates the Globally asynchronous Locally Synchronous (GALS) design style, creating an asynchronous network-on-chip (ANoC). Synchronizing FIFOs are critical components to interface between the ANoC and the functional domains as depicted in Figure 1 (left).

As an example, Figure 1 (right) shows a simplified view of a modern multi-core CPU. In this figure, each core has its own L1 and L2 (level-1 and level-2) caches, and an on-chip network connects the cores to a shared L3 cache and accelerator. The CDC (clock-domain crossing) boxes provide the interfaces between different timing domains. If the cores, NoC (network-on-chip), L3 caches, and accelerator each operate with their own clocks, then each CDC module must include a synchronizer, and the synchronization latency is added to the total latency of the data transfer. The example in Figure 1 (right) shows that four such clock-domain crossings are used to handle an L2 cache miss. With core-clock frequencies of 3GHz or more, three-flip-flop synchronizers are common, and the synchronization alone can contribute twelve cycles to the total miss-processing time. Architects are always asking for higher NoC bandwidth with lower latency, and a 12 cycle synchronization penalty is a significant performance issue.

For these problems, asynchronous solutions offer several advantages. First, no synchronization is needed when entering the asynchronous time-domain. For the example of the multi-core CPU, by simply using an ANoC [2]–[6], the twelve cycle synchronization penalty of the all-synchronous example design can be alleviated to half that, *i.e.*, six cycles.

It may also be advantageous that CPUs or accelerators be asynchronous. There are several asynchronous design languages that support the design of asynchronous blocks from scratch [7]–[11]. Alternatively, desynchronization is an attractive approach that leverages existing synchronous RTL specifications, commercial synthesis tools, and cell libraries

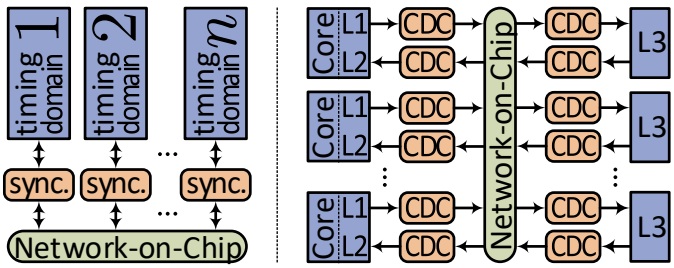

Fig. 1: (left) A Network-on-Chip accessing multiple timing domains. (right) An example of a multi-core CPU crossing clock domains to access caches.

Many thanks to the anonymous reviewers for their feedback and suggestions. An earlier version of this work appeared in the 2020 13th International Workshop on Network on Chip Architectures (NoCArc) [1].

978-1-7281-9226-0/20/$31.00 © 2020 IEEE

to generate asynchronous bundled-data pipelines [12], [13]. These pipelines use local clocks generated via asynchronous control, thus they are resilient to process, voltage, and temperature (PVT) variations [14], [15] and benefit from dynamic voltage scaling. As in ANoCs, the integration with low-latency synchronizers is critical for making these systems usable.

This paper thus focuses on a family of interfaces for crossing between synchronous and asynchronous timing domains. Our asynchronous interfaces assume the Asynchronous Symmetric Persistent Pulse Protocol (asP*) [16] because wire-delay is a key performance limiter for large blocks or blocks that span a large portion of a chip (*e.g.*, a NoC). Using the asP* protocol, each data transfer requires a minimal pulse width to travel a round-trip between the sender and receiver: first, data and a request minimum pulse are sent from the sender to the receiver; in response, the receiver sends an acknowledgement minimum pulse back to the sender. The throughput of the network is limited by this round-trip time. While a two-phase protocol doesn't require a return-to-zero transition, it's implementation is more complicated. On the other hand, if a four-phase protocol is used, two round-trips are required for each data transfer, achieving roughly half the throughput of a two-phase design. The asP* protocol benefits from the simplicity of return-to-zero protocols, while achieving high speeds due to minimum-pulse communication.

By providing a synthesizable library of high-throughput, low-latency timing-domain crossing interfaces, we provide designers with a disciplined way to incorporate the use of ANoCs and asynchronous modules into designs where some or many modules are designed and synthesized using a traditional, synchronous design flow. We believe these interfaces greatly lower the barrier to entry for exploiting the advantages of asynchronous designs in a heterogeneous design framework. Our timing-domain crossing FIFOs are fully synthesizable using standard-cell-based libraries, highly configurable, and support any combination of synchronous and asynchronous interfaces. The designs are highly parameterized and synthesizable using standard, commercial cell libraries and design flows. The complete design framework is public and open-source, including a Verilog description of the FIFOs, support for simulation, static timing analysis, a testbench for regression tests of the design, and a run-in-batch flow manager [17].

The remainder of this paper is organized as follows. Section II reviews related work. Section Section III presents our proposed synchronizing FIFOs. In Section IV our experimental framework is presented and experimental results are discussed. Throughput, latency, and other FIFO metrics are generated with our open-source Verilog and standard design tools using FreePDK45 [18], [19], a cell library for a 45nm process. Our FIFOs achieve throughputs that exceed typical clock frequencies for ASIC design flows. Finally, Section V concludes the paper with future suggestions.

II. Background, Preliminaries and Related Work

With the rapid progress of silicon technology, modern devices comprise an increasing number of on-chip design modules, incorporate multiple clock domains running at higher frequencies. As these design challenges hinder system integration and timing closure, FIFOs become a more attractive solution for decoupling and transferring data between different domains because they offer high throughput and simple flow control. Synchronizing FIFOs are largely distinguished by the design of the interface control logic, the data storage, and the synchronization mechanisms between the interfaces.

Gray-code FIFOs. The most common synchronizing FIFOs are based on Gray-code counters [20], [21]. The advantage of a Gray-code is that on a clock transition, exactly one bit of the counter makes a transition. If the put-controller uses a Gray-code for its write pointer, then the bits of the pointer can be synchronized to the get-controller using a separate synchronizer for each bit. Because at most one bit will be changing at the receiver (get) interface clock edge, at most one synchronizer will enter metastability. When that bit resolves, the synchronizer outputs either the "before" or "after" value of the write-pointer. Either is valid. The disadvantage of Gray-codes is the difficulty of comparing two Gray-code values to determine which is greater. Typical designs convert the Gray-code value to binary, and then perform the comparison. The conversion requires a chain of XOR gates whose length is the number of bits in the pointer (minus one). This tends to be a slow operation that limits FIFO performance.

Unary-code FIFOs. An alternative to Gray code pointers is to use some kind of unary encoding. Like several other designs [22]–[25], our approach uses a unary encoding of the FIFO pointers. These FIFOs offer very high throughputs because ring counters are fast, and comparing unary values is easy. Of these, our design is standard-cell-based. The main disadvantage of unary control is the larger flip-flop count, especially for the synchronizers. For desynchonization applications, FIFO depths tend to be small whereas the word-width tends to be fairly high. Both of these properties mitigate the overhead of using unary control.

Pausible clock FIFOs. In addition to Gray code and unary counters, many other designs have been proposed. Keller [26] presents a novel implementation for GALS applications based on "pointer-increment" signals. Keller's design uses mutex elements to arbitrate between communication and clock generation; because metastability is rare and usually resolves quickly, Keller's design, like most pausible clock designs, achieves very low latency for cross-domain communication. While we note a growing interest in pausible clock GALS (*e.g.*, [20], [26]), the most common clock-domain-crossing designs remain synchronous-to-synchronous.

Ripple FIFOs. Another approach to synchronization is to use a ripple FIFO instead of a pointer based design. Seizovic [27] showed synchronization can be incorporated into the control path of a ripple FIFO. More recently, Jackson and Manohar [28] showed a generalization of Seizovic's scheme where a pipeline processing can be done along the datapath of the FIFO while the control path accomplishes synchronization. These are clever designs, but they use special handshaking cells that are not amenable for standard synthesis flows.

Mesochronous FIFOs. Finally, there has been extensive work on mesochronous designs where the sender and receiver operate at identical clock frequency with an unknown phase relationship [29]–[33]. When the communicating clock domains have a common source, these methods offer excellent performance and efficiency. We are addressing the more general, and common case, where either there is no common source to the clocks, or where it is impractical to bound the variation in the skew under operation. We note that [33] provides an excellent survey of prior work in clock-domain crossing that transcends the space limitations of this paper.

III. The FIFO Architecture

Our goal is to support the design of asynchronous drop-in replacements for synchronous blocks with latency-insensitive interfaces. This enables incremental incorporation of asynchronous blocks for functions where they provide advantages as well as an interface to chip-wide asynchronous NoCs. From the synchronous designer's perspective, the asynchronous module(s) communicate through latency-insensitive interfaces; and no special considerations are needed to account for the asynchronous implementation on the other side of these interfaces. The two key interfaces are a synchronous-to-asynchronous converter that transfers data from the clocked, synchronous domain to an asynchronous module using the asP* protocol. The asynchronous-to-synchronous converter is the inverse: it transfers data, with proper synchronization, from the asynchronous timing domain back to the clocked domain. The modular design of our interfaces naturally provides synchronous-to-synchronous converters that transfer data between two synchronous domains with independent clocks. While the same modularity also allows the construction of an asynchronous-to-asynchronous interface, such interfaces are rarely, if ever, needed. Unlike synchronous designs, asynchronous modules are naturally composable without imposing timing-closure headaches on their interfaces.

The FIFO rings. Figure 2 shows the structure of our synchronizing FIFO. A Sender in *timing domain A* communicates with the Receiver in *timing domain B* through the FIFO. The FIFO is composed of two round-robin rings of n stages, the *put ring* and the *get ring*. Each stage has a put interface cell, a get interface cell and a full-empty control. In addition, each FIFO stage might include a data store. Alternatively, a separate two-ported memory array may be used, with control signals coming from the put and get interface cells.

The FIFO uses *tokens* to mark the location of the next write and read operation. Initially, the put and get tokens are in the put and get interface cells of stage 1. Each time a data value is written to the FIFO, the put token is advanced to the next stage. Similarly, the get interface cells advance the get token on a read operation. The structure of a FIFO stage is shown in Figure 3. For simplicity, only some signals are shown. Next sections give a more detailed account of all signals involved in the operation. Signals `put_token_in` and `get_token_in` come from the previous stage, while `put_token_out` and `get_token_out` go to the next stage. Initially the stage is empty and signal stage `full` is low and stage `empty` is high. If signal `put_token_in` is high, an incoming `req_put` causes signal `write` to go high, which in turn causes the full-empty control to raise stage `full` and lower stage `empty`. After a successful write operation, the token is transferred from this stage to the next by lowering `put_token_in` and raising `put_token_out`. Likewise, a `req_get` request causes signal `read` to go high and subsequently stage `full` to drop and stage `empty` to go high. In the data store, signal `write` causes the input data `datain` to be stored, while signal `read` causes the output data bus `dataout` to be driven with the data value for this stage. This will be explained in more detail in next sections. There are two types of put and get interface cells, clocked and clockless. Similarly, the full-empty control can be clocked or clockless.

A. Full-Empty Control

Each FIFO stage has one full-empty control block. Figure 4 shows the full-empty control block, consisting of three portions. The put interface (left side), the synchronization block (middle), and the get interface (right side). The put/get interface can be either clocked or clockless. Synchronization is only required when crossing between clocked interfaces. Clocked and clockless interfaces can be combined in all possible mix-and-match ways, such as two clocked interfaces (upper left case), or two clockless interface (lower right case). In the clockless put interface, the `write` signal causes the toggling of flop f_{ap}, while in the clocked get interface the flop f_{sg} toggles on a rising read clock edge, when signal `read` is high. The two flip-flops f_{ap} and f_{sg} encode the state of the FIFO stage: if the outputs of the flip-flops are the same, then the FIFO stage is empty; if they are different, then the FIFO is full. The clocked put/get interface requires a synchronizer to minimize metastability-related failure. The synchronizer can consist of any number of half-cycle and full-cycle synchronization stages. Note that only the signal from the other clock domain needs to pass through the synchronizer. Thus, a state change due to a read operation causes the stage `empty` signal to go high without incurring the synchronization latency penalty, and similarly for write operations and stage `full`. As a result, the FIFO won't overflow or underflow.

Distributed synchronization. As described above, our FIFO has a one-bit synchronizer in each clocked full-empty interface. This is in contrast to traditional synchronizing FIFO

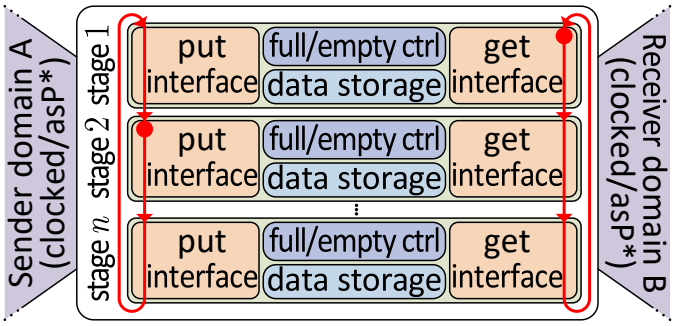

Fig. 2: Architecture overview.

designs that have a top-level synchronizer. We chose to use a distributed synchronization scheme to make our design more modular. If the synchronized full signal is asserted in the receiver's domain, then the stage is guaranteed to hold data and therefore that a read is safe. Likewise, assertion of the synchronize empty signal in a clocked sender's domain guarantees that the stage is empty and that a write to the stage is safe. This design prevents invalid states from being sampled, since it is acceptable for any changing bit to have its old or new value sampled.

B. A Simplified asP FIFO*

The Asynchronous Symmetric Persistent Pulse Protocol (asP*) [16] is used to implement our asynchronous interfaces because it offers major benefits compared to other asynchronous protocols. First, asP* exhibits high speeds, enabling low latencies and high performance. Also, the asP* can be implemented using common standard-cell-libraries and doesn't require special cells such as C-elements.

A simplified asP* FIFO that is the base of our design is shown in Figure 5. This FIFO consists of a control and datapath portions. The datapath is a chain of D-latches that shifts the data from `datain` towards `dataout`. This datapath is controlled by an SR-latch-based control circuit. Each SR-latches in the controller indicated that its corresponding stage is full, namely the D-latch in the same stage holds a valid data. For each stage, the AND gate indicated that the data should be moved to the current stage if the previous stage is full AND the current stage is empty. For stage i, if the previous stage is full, $Q(SRL_{i-1}) = `1'$, and the current stage is empty $\bar{Q}(SRL_{i-1}) = `1'$, DL_i will be enabled and data will move from stage $i-1$ to stage i. Also, the reset signal (R) of SRL_{i-1} will be asserted indicating that stage $i-1$ is now empty, and the set signal (S) of SRL_i will be asserted, indicating that stage i is now full.

Timing constraints of the simplified asP FIFO.* As the asP* is pulse-based, it is not delay-insensitive. Once a D-latch DL_i is enabled, the same enable signal (AND's output) will reset SRL_{i-1} and set SRL_i concurrently. This will lower the output of the same AND gate and de-assert the enable signal of DL_i, creating a pulse on the enable. This pulse is supposed to be longer than minimum pulse constraint of a D-latch, namely,

$$\min\begin{Bmatrix}\text{minDelay(SRL.R}\rightarrow\text{Q)},\\ \text{minDelay(SRL.S}\rightarrow\bar{\text{Q}})\end{Bmatrix}\geq\text{minPulseWidth(DL.EN)}. \quad (1)$$

Another difficulty presented by the asP* FIFO shown in Figure 5 is the use of SR-latches as state-holding elements. Many standard-cell libraries do not include SR latches. Although SR latches can be effected by using cross-coupled NAND or NOR gates or by using flip-flops with asynchronous set and reset inputs, such designs are not amenable to static timing analysis by standard CAD tools that assume timing paths start and end according to the clock inputs of flip-flops. The FIFO that we present in the remainder of this section implements the asP* FIFO, but does so with a design that avoids these problems.

C. The FIFO Interface

The FIFO's top-level signals and stage connectivity are shown Figure 6. A put request from the sender is broadcast to all FIFO stages. In the clocked version the put request will be broadcasted if there is an available space, thus it will be masked with the `space_avail` signal. The put acknowledge signals `ack_put` from all FIFO stages are combined in an OR tree to the acknowledge signal back to the sender. Similarly, the clocked version is the `space_avail` signal where all `wrt_enb` signals from all stages are OR'ed. At any time a maximum of one stage will raise its `ack_put`/`wrt_enb` signal. A clockless received will acknowledge a get request, `req_get`, by raising the signal `ack_get`, if any one of the stages acknowledges the get by raising its `ack_get` signal. On the clocked receiver side, the FIFO tells the receiver that it has valid data at its `dataout` output by raising signal `data_v`, which is the OR'ed `read_enb` signals.

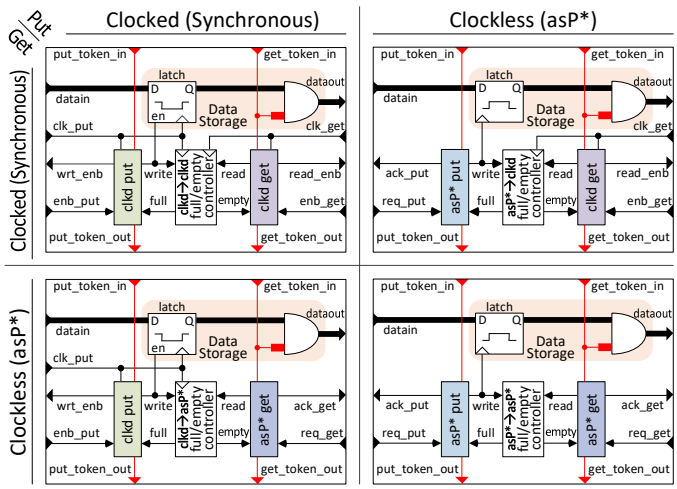

Fig. 3: A single FIFO stage for all mixed-timing combinations.

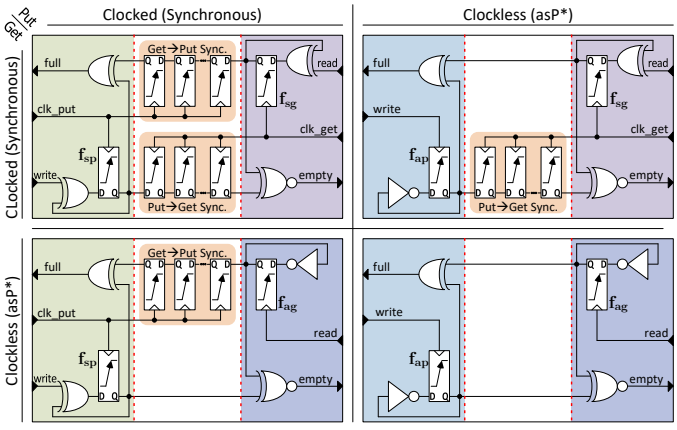

Fig. 4: Full/empty controller for all mixed-timing combinations.

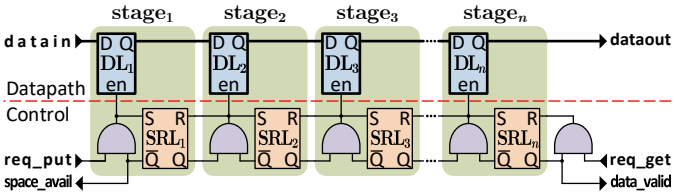

Fig. 5: Asynchronous Symmetric Persistent Pulse Protocol (asP*) pipeline.

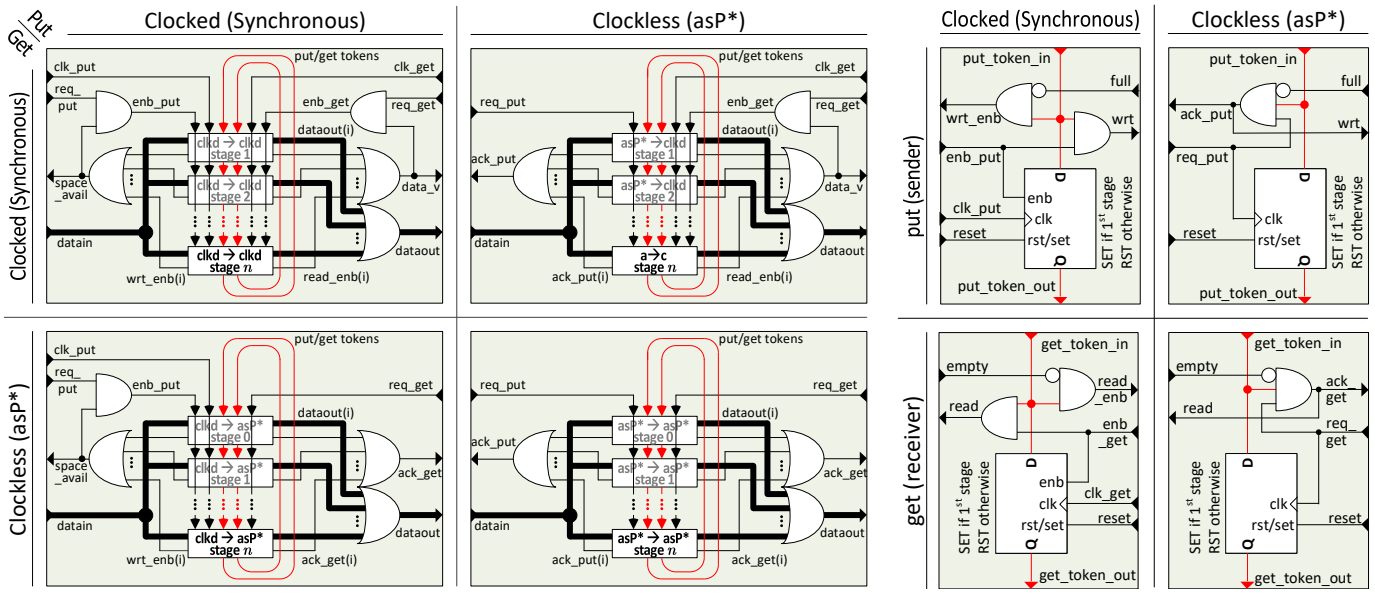

Fig. 6: The FIFO structure for all mixed-timing combinations.

Fig. 7: Interface components.

The FIFO interface units (within each stage) are shown in Figure 7. The get and put rings are part of the get and put interfaces, respectively. Each ring is composed of a shift register (one flip-flop per stage) holding a one-hot value. Initially, the token is located in the first stage, thus the first flip-flop stores 1, while all other flip-flops are zero. For the clocked interface, the token ring moves forward on the clock edge, and is enabled by `enb_put/enb_get`, for the put/get interfaces, respectively. The clockless ring, on the other hand, moves on the rising edge of `req_put/req_get`, for the put and get interfaces, respectively. Other signals are generated based on the token ring, which indicated the current active stage for get and put interfaces. In the clocked interfaces, `wrt/read` signals are asserted if the current stage is active, and there is a put/get request, respectively. `wrt_enb/read_enb` are asserted if the current stage is active, and the current stage is not full/empty, respectively. The clockless interface behaves similarly, however, `req_put/req_get` signals are incorporated in the generation of `wrt/read` signals, respectively.

Minimum pule-width timing constraints. To ensure the correct functionality of the asP* FIFO, the following minimum-pulse width must be satisfied by the asP* interface user. `req_put/req_get` signals are used to trigger the put/get token ring flip-flop, and the f_{ap}/f_{ag} flip-flops in the full-empty control, respectively, thus,

$$\text{minPulseWidth}(\text{req\_put/get}_{\text{LO}}) \geq \text{minPulseWidth}(\text{DFF.CLK}_{\text{HI}}). \quad (2)$$

All FFs are triggered simultaneously to pass on the put and get tokens, and the propagation delay from one flip-flop to the next has to be greater than the hold time of the flip-flop; otherwise additional delay needs to be inserted into the token wires to ensure that the hold time constraints on the flip-flops are met. Standard timing tools can be used to check and fix these timing constraints as described in Section IV.

D. Data Store

To reduce storage area, latches are used to store data instead of registers as described in Figure 3. Each FIFO stage has a latch array for the data; the latch enable is controlled by the stage's write signal. Incoming data is broadcast to each FIFO stage, and data latches in stage i are transparent when stage i's write signal is high. At any given time, only one stage will have a high write signal and only one stage will store the incoming data. For FIFO read operations, the get token determines which FIFO stage drives the output data. The selection can be done with pass-gates, tri-state buffers, or multiplexers. For example, the design described in Figure 6 uses gate-based multiplexers for ease of implementation with a standard CAD design flow. The writing and reading with clocked interfaces is straightforward. The `write` signal is used to enable the storage latches. Note that `write` is only high during the clock low phase, such that the write clock high phase can be used to drive the data onto the bit lines, whenever `enb_put` is asserted. A read operation is performed when `enb_get` is asserted. Signal `read` is only high for half the cycle, leaving the first half of the read clock cycle for precharge. The data can be gated using the get token.

IV. Experimental Results

Platform Settings. In order to verify and simulate the suggested approach, we wrote a fully parameterized Verilog module. To simulate and synthesize the proposed designs with various parameters in batch using Synopsys Synthesis tools [34]–[38], we also provide a run-in-batch flow manager. The Verilog modules and the flow manager are available online [17]. The design package is composed of a Verilog description of the proposed FIFO, a Verilog testbench, tool configuration scripts, and a run-in-batch manager. The Verilog description of the proposed FIFO is parameterized and can be instantiated in other Verilog projects as a stand-alone IP.

The complete design framework is illustrated in Figure 8. A run-in-batch manager allows synthesizing and simulating a number of FIFOs in batch. A list of design parameters (*e.g.*, interface protocols,stages, data width, and synchronizers' depth) can be provided to the run-in-batch manager, together with the required flow stages. The design stages are implemented using Synopsys tools and ordered as follows. (1) Logic synthesis using Design Compiler [34]. (2) Placement and routing using IC Compiler [35]; this include delay and parasitic extraction [36]. (3) Static timing analysis using PrimeTime [37]. (4) Gate-level-Simulation using VCS [38] with the placed and routed netlist, delays and parasitics data, and the Verilog testbench. The testbench generates input vectors, checks the outputs, and compares them against a generic FIFO. The simulation also generates the activity data for power analysis. (5) Power analysis using PrimeTime PX [39] to estimate both dynamic and leakage power.

Experimental Results. Our FIFOs are implemented using FreePDK45, an open-source 45nm CMOS standard-cell library [18], [19]. The performance of each FIFO configuration is shown in Table I. For the clocked FIFOs, this is the maximum clock rate at which the circuits can operate, while for the asynchronous FIFOs this is the fastest rate at which requests can be serviced. The worst-case timing library was used for all measurements. The data in Table I shows that the throughput of the FIFO scales relatively well as the number of stages is increased from 8 to 32. Both clocked and asynchronous FIFOs are able to run at over 1.92 GHz with 32 stages. All FIFOs have a data throughput equivalent to the maximum clock/request frequency.

The minimum latency through the FIFOs is also listed in Table I. These were measured from timing-annotated simulations using values automatically extracted from the placed and routed design. All numbers assume worst-case process, voltage and temperature. For the synchronous FIFOs we assumed both clocks had the maximum clock frequency and were in phase; then the minimum latency is simply 5 clock cycles.

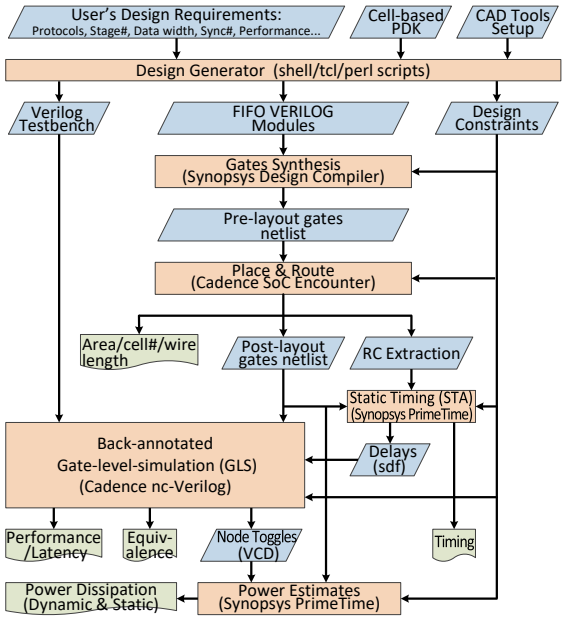

Fig. 8: Design framework.

TABLE I: Resources Consumption of Multiple FIFO Instances.

FIFO Configuration			Throughput	Latency	Area	Data	Power
Put	Get	Stages	Gtransfers/s	[ns]	[μm^2]	%	[μW]
clkd	clkd	8	2.19	2.96	2513.82	46.74	66.48
clkd	clkd	16	2.14	3.23	3760.59	50.33	91.20
clkd	clkd	32	1.97	3.55	8490.22	54.55	127.68
asP*	clkd	8	2.27	2.86	1674.59	60.62	65.78
asP*	clkd	16	2.23	3.02	3117.22	61.43	126.50
asP*	clkd	32	1.93	3.20	6001.06	68.21	199.85
clkd	asP*	8	2.35	2.89	1733.51	63.03	48.51
clkd	asP*	16	2.2	2.98	3004.55	64.26	90.95
clkd	asP*	32	1.92	3.25	6953.87	67.41	212.76
asP*	asP*	8	2.29	0.35	1247.34	80.14	38.08
asP*	asP*	16	2.32	0.43	2586.88	81.26	133.25
asP*	asP*	32	2.03	0.48	5908.89	83.32	213.56

The asynchronous FIFO uses significantly less area than the synchronous FIFO, owing to the fact the asynchronous implementation does not require synchronizing flip-flops. Over 80% of the asynchronous FIFO area is taken up by the latch-based data store, while for the clocked FIFOs the data uses as little as 46% and as much as 54% of the total area.

The power analysis was based on a simulation with 100k random transactions, which on average generated a new transaction for 50% of the available slots. Each simulation was done with the same 100k random transactions and with the same operating frequency. The power numbers appear to reflect that the place and route tools are not optimizing for power. Once the critical paths are optimized for speed, other cells can be placed nearly arbitrarily due to the small size of the FIFO. However, this creates a large variation on the internal wire loads that are driven by the different FIFOs.

V. Conclusions and Future Work

This paper presents modular, high-performance synthesizable interfaces for crossing between asynchronous and synchronous timing domain. Our FIFO is available as an open-source, highly parameterized design [17]. We provide scripts for an industry standard design flow based on Synopsis tools. This includes synthesis, layout generation, extraction, timing analysis, simulation, and power estimation. We include a simulation testbench, and scripts for batching sweeping design parameters to enable exploration of the design space. Our design should be both useful to designers and provide a repeatable reference case for other researchers.

Our design exclusively uses cells that are available in a typical standard-cell library, making it well-suited for use in an ASIC design flow. In particular, we presented implementations of this design using the FreePDK, an open-source 45nm CMOS standard-cell library [18], [19] and a standard ASIC CAD flow. Even without the use of full-custom logic, all of our FIFOs achieve throughput exceeding 1.92 giga-transfer per second for both synchronous and asynchronous version.

As a future work, we are planning to support SRAM-based storage instead of latch-based storage to increase area efficiency [40]–[42]. Intentional clock skew can be utilized to increase the performance of the SRAM blocks [43], [44]. Furthermore, the two-phase protocol can be supported.

REFERENCES

[1] A. M. S. Abdelhadi, "High-Throughput Synthesizable Synchronization FIFOs for Mixed-Timing NoCs," in *Proceedings of the 13th International Workshop on Network on Chip Architectures (NoCArc)*, Oct. 2020.

[2] D. Lattard *et al.*, "A Reconfigurable Baseband Platform Based on an Asynchronous Network-on-Chip," *IEEE J. of Solid-State Circuits (JSSC)*, vol. 43, no. 1, pp. 223–235, Jan. 2008.

[3] D. Gebhardt, J. You, and K. S. Stevens, "Design of an Energy-Efficient Asynchronous NoC and Its Optimization Tools for Heterogeneous SoCs," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Syst. (TCAD)*, vol. 30, no. 9, pp. 1387–1399, Sept. 2011.

[4] T. Bjerregaard and J. Sparso, "Implementation of Guaranteed Services in the MANGO Clockless Network-on-Chip," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 4, pp. 217–229, July 2006.

[5] R. Dobkin, R. Ginosar, and I. Cidon, "QNoC Asynchronous Router with Dynamic Virtual Channel Allocation," in *Proc. of the IEEE/ACM Int. Symp. on Networks-on-Chip (NOCS)*, May 2007, pp. 218–218.

[6] J. Bainbridge and S. Furber, "Chain: A Delay-Insensitive Chip Area Interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, Sept. 2002.

[7] K. van Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, "The VLSI-Programming Language Tangram and Its Translation Into Handshake Circuits," in *Proc. of the European Conf. on Design Autom.*, 1991, pp. 384–389.

[8] D. Edwards and A. Bardsley, "Balsa: An Asynchronous Hardware Synthesis Language," *The Computer Journal*, vol. 45, no. 1, pp. 12–18, 2002.

[9] S. F. Nielsen, J. Sparsø, J. B. Jensen, and J. S. R. Nielsen, "A behavioral synthesis frontend to the haste/tide design flow," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2009, pp. 185–194.

[10] A. Bardsley, L. Tarazona, and D. Edwards, "Teak: A Token-Flow Implementation for the Balsa Language," in *Proc. of the Int. Conf. on the Appl. of Concurrency to Syst. Design (ACSD)*, July 2009, pp. 23–31.

[11] M. Renaudin and A. Fonkoua, "Tiempo Asynchronous Circuits System Verilog Modeling Language," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2012, pp. 105–112.

[12] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Syst. (TCAD)*, vol. 25, no. 10, pp. 1904–1921, Oct. 2006.

[13] N. Andrikos, L. Lavagno, D. Pandini, and C. P. Sotiriou, "A Fully-Automated Desynchronization Flow for Synchronous Circuits," in *Proc. of the Annu. Design Autom. Conf.*, June 2007, pp. 982–985.

[14] J. Cortadella, M. Lupon, A. Moreno, A. Roca, and S. S. Sapatnekar, "Ring Oscillator Clocks and Margins," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2016.

[15] Y. Zhang, H. Zha, V. Sahir, H. Cheng, and P. Beerel, "Test Margin and Yield in Bundled Data and Ring-Oscillator Based Designs," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2017, pp. 85–93.

[16] C. E. Molnar, I. W. Jones, W. S. Coates, J. K. Lexau, S. M. Fairbanks, and I. E. Sutherland, "Two FIFO Ring Performance Experiments," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 297–307, 1999.

[17] A. M. S. Abdelhadi, GitHub Open-Source Repository. [Online]. Available: https://github.com/AmeerAbdelhadi/cell-based_mixed_fifo.flow

[18] J. E. Stine *et al.*, "FreePDK: An Open-Source Variation-Aware Design Kit," in *Proc. of the IEEE Int. Conf. on Microelectronic Systems Education (MSE)*, June 2007, pp. 173–174.

[19] J. E. Stine *et al.*, "FreePDK v2.0: Transitioning VLSI Education Towards Nanometer Variation-Aware Designs," in *Proc. of the IEEE Int. Conf. on Microelectronic Systems Education (MSE)*, Sept. 2009, pp. 100–103.

[20] R. W. Apperson, Z. Yu, M. J. Meeuwsen, T. Mohsenin, and B. M. Baas, "A Scalable Dual-Clock FIFO for Data Transfers Between Arbitrary and Haltable Clock Domains," *IEEE Trans. on Very Large Scale Integration Syst. (TVLSI)*, vol. 15, no. 10, pp. 1125–1134, Oct. 2007.

[21] C. Cummings and P. Alfke, "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons," in *Synopsys Users Group Conference (SNUG)*, Jan. 2002.

[22] T. Chelcea and S. M. Nowick, "Robust Interfaces for Mixed-Timing Systems," *IEEE Trans. on Very Large Scale Integration Syst. (TVLSI)*, vol. 12, no. 8, pp. 857–873, Aug. 2004.

[23] I. M. Panades and A. Greiner, "Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures," in *Proc. of the IEEE/ACM Int. Symp. on Networks-on-Chip (NOCS)*, May 2007, pp. 83–94.

[24] T. Ono and M. Greenstreet, "A Modular Synchronizing FIFO for NoCs," in *Proc. of the IEEE/ACM Int. Symp. on Networks-on-Chip (NOCS)*, May 2009, pp. 224–233.

[25] A. M. S. Abdelhadi and M. R. Greenstreet, "Interleaved Architectures for High-Throughput Synthesizable Synchronization FIFOs," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2017, pp. 41–48.

[26] B. Keller, M. Fojtik, and B. Khailany, "A Pausible Bisynchronous FIFO for GALS Systems," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2015, pp. 1–8.

[27] J. N. Seizovic, "Pipeline Synchronization," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, Nov. 1994, pp. 87–96.

[28] S. Jackson and R. Manohar, "Gradual Synchronization," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2016, pp. 29–36.

[29] D. G. Messerschmitt, "Synchronization in Digital System Design," *IEEE J. Sel. Areas Commun (J-SAC)*, vol. 8, no. 8, pp. 1404–1419, Oct. 1990.

[30] M. R. Greenstreet, "STARI: A Technique for High-Bandwidth Communication," Ph.D. dissertation, Department of Computer Science, Princeton University, Jan. 1993.

[31] A. Chakraborty and M. R. Greenstreet, "Efficient Self-Timed Interfaces for Crossing Clock Domains," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2003, pp. 78–88.

[32] W. J. Dally and S. G. Tell, "The Even/Odd Synchronizer: A Fast, All-Digital, Periodic Synchronizer," in *Proc. of the IEEE Int. Symp. on Asynchronous Circuits and Syst. (ASYNC)*, May 2010, pp. 75–84.

[33] D. Verbitsky, R. R. Dobkin, R. Ginosar, and S. Beer, "StarSync: An Extendable Standard-cell Mesochronous Synchronizer," *Integr. VLSI J.*, vol. 47, no. 2, pp. 250–260, Mar. 2014.

[34] *Design Compiler User Guide*, Synopsys Inc., Mar. 2016, ver. L-2016.03-SP1.

[35] *IC Compiler Implementation User Guide*, Synopsys Inc., Mar. 2016, ver. L-2016.03-SP4.

[36] *StarRC User Guide and Command Reference*, Synopsys Inc., Dec. 2015, ver. K–2015.12.

[37] *PrimeTime User Guide*, Synopsys Inc., Dec. 2015, ver. K–2015.12.

[38] *VCS MX/VCS MXi User Guide*, Synopsys Inc., May 2016, ver. L–2016.06.

[39] *PrimeTime PX User Guide*, Synopsys Inc., Dec. 2016, ver. M–2016.12.

[40] A. M. S. Abdelhadi and G. G. F. Lemieux, "A Multi-ported Memory Compiler Utilizing True Dual-Port BRAMs," in *Proc. of the IEEE Annu. Int. Symp. on Field-Programmable Custom Comput. Mach. (FCCM)*, Aug. 2016, pp. 140–147.

[41] A. M. S. Abdelhadi and G. G. F. Lemieux, "Deep and Narrow Binary Content-Addressable Memories Using FPGA-Based BRAMs," in *Proc. of the Int. Conf. on Field-Programmable Technol. (FPT)*, Dec. 2014, pp. 318–321.

[42] A. M. S. Abdelhadi, G. G. F. Lemieux, and L. Shannon, "Modular Block-RAM-Based Longest-Prefix Match Ternary Content-Addressable Memories," in *Proc. of the Int. Conf. on Field Programmable Logic and Applications (FPL)*, Aug. 2018, pp. 243–2437.

[43] A. Brant, A. Abdelhadi, A. Severance, and G. G. F. Lemieux, "Pipeline Frequency Boosting: Hiding Dual-Ported Block RAM Latency Using Intentional Clock Skew," in *Proc. of the Int. Conf. on Field-Programmable Technol. (FPT)*, Dec. 2012, pp. 235–238.

[44] A. Brant, A. Abdelhadi, D. H. H. Sim, S. L. Tang, M. X. Yue, and G. G. F. Lemieux, "Safe Overclocking of Tightly Coupled CGRAs and Processor Arrays using Razor," in *Proc. of the IEEE Annu. Int. Symp. on Field-Programmable Custom Comput. Mach. (FCCM)*, June 2013, pp. 37–44.