2021 31st International Conference on Field-Programmable Logic and Applications (FPL) | 978-1-6654-3759-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/FPL53798.2021.00062

Enabling Mixed-Timing NoCs for FPGAs: Reconfigurable Synthesizable Synchronization FIFOs

Ameer M. S. Abdelhadi
University of Toronto
Toronto ON M5S 3G4, Canada
ameer.abdelhadi@utoronto.ca

He Li
University of Cambridge
Cambridge CB2 1TN, United Kingdom
he.li@ieee.org

***Abstract*—We present an architecture of a reconfigurable high-throughput synthesizable synchronization FIFO for crossing between asynchronous and synchronous timing domains. This FIFO is composed of reconfigurable 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. The FIFO is fully synthesizable using widely available standard-cell libraries and a standard ASIC design flow. Our post-layout design can operate at speeds greater than 1.2 giga-transfer per second under worst-case conditions when implemented in a 65nm CMOS process.**

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

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 [1], 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. The CDC (clock-domain crossing) boxes provide the interfaces between different timing domains. If the cores, NoC (network-on-chip), and L3 caches 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 12 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]–[5], the twelve-cycle synchronization penalty of the all-synchronous design can be alleviated to half that, *i.e.*, six cycles.

This paper thus focuses on a reconfigurable interface for crossing between synchronous and asynchronous timing domains. Our asynchronous interface assume the Asynchronous Symmetric Persistent Pulse Protocol (asP*) [6] 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). As depicted in Figure 2, each data transfer of the asP* protocol requires a minimal width pulse 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. For each transmission, each signal returns to zero, however, the return transition is done immediately after a minimum pulse. A four-phase protocol is also a return-to-zero (RZ), but requires two round-trips for each data transfer, achieving roughly half the throughput of a two-phase of an asP* design. While a two-phase protocol does not require a return-to-zero transition (NRZ), its implementation is more complicated. The asP* protocol benefits from the simplicity of return-to-zero protocols, while achieving high speeds due to minimum-pulse communication.

By providing a reconfigurable and synthesizable design of high-throughput, low-latency timing-domain crossing interface, we provide FPGA designers with a disciplined way to incorporate the use of ANoCs and asynchronous modules into their designs. 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 FIFO is fully synthesizable

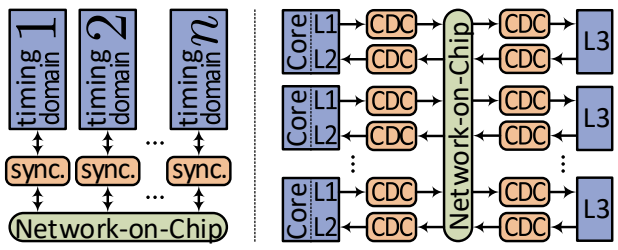

Fig. 1. (left) A Network-on-Chip accessing multiple timing domains. (right) An example of a multi-core CPU crossing clock domains to access caches.

Many thanks to the anonymous reviewers for their feedback and suggestions. This research would not have been possible without the access to design tools and libraries that have been obtained through the Canadian Microelectronics Corporation (CMC). Corresponding author: He Li.

978-1-6654-3759-2/21/$31.00 ©2021 IEEE
DOI 10.1109/FPL53798.2021.00062

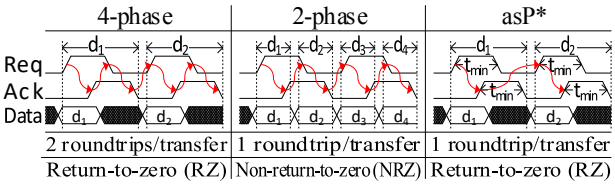

Fig. 2. Self-timed handshake protocols.

using standard-cell-based libraries and standard design flows, highly configurable, and supports any combination of synchronous and asynchronous interfaces. The complete design framework is public and open-source, including a Verilog description of the FIFO, support for synthesis and simulation, a testbench for regression tests, and a run-in-batch flow manager [7].

The remainder of this paper is organized as follows. Synchronizing FIFOs are reviewed in Section II. Section III presents our proposed synchronizing FIFOs. Section IV provides an analysis of throughput and latency. In Section V our experimental framework is presented, results are discussed, and other approaches are compared. Finally, Section VI concludes the paper with future suggestions.

II. Background, Preliminaries and Related Work

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 for both ASICs [8] and FPGAs [9], [10], are based on Gray-code counters. 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-codes 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 [11], [12], 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 requirement of large flip-flop counts, 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 [13] 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.*, [8], [13]), the most common clock-domain-crossing designs remain synchronous-to-synchronous, and we focus on that scenario here.

Ripple FIFO. Another approach to synchronization is to use a ripple FIFO instead of a pointer based design. Seizovic showed synchronization can be incorporated into the control path of a ripple FIFO [14]. More recently, Jackson and Manohar showed a generalization of Seizovic's scheme where some pipeline processing can be done along the datapath of the FIFO while the control path accomplishes synchronization [15]. However, they use special handshaking cells that are not amenable for standard synthesis.

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 modules 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 (S2A) converter that transfers data from the clocked, synchronous domain to an asynchronous module using the asP* protocol. The asynchronous-to-synchronous (A2S) 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 (A2A) 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 configuration fields of our proposed FIFO are listed in Table I.

A. A Simplified asP* FIFO

The Asynchronous Symmetric Persistent Pulse Protocol (asP*) [6] 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. Using the asP* protocol, each data transfer requires a minimal width pulse 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 does not require a return-to-zero transition, its implementation is more complicated. On the other hand, a four-phase protocol requires two round-trips 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 and achieves high speeds due to minimum-pulse communication. The asP* can also be implemented using standard-cell-libraries and does not require special cells such as C-elements.

TABLE I
FIFO Configuration Fields.

Field	#Bits	Description	Possible values
PP	1	Put Protocol	'0': asP*, or '1': clocked
GP	1	Get Protocol	'0': asP*, or '1': clocked
NFFS	2	#FFs in Synchronizer	'01': 1, '10': 2, or '11': 3 FFs

A simplified asP* FIFO that is the base of our design is shown in Figure 3. 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-latch in the controller indicates that its corresponding stage is full, namely the D-latch in the same stage holds a valid data. For each stage, the AND gate indicates 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.

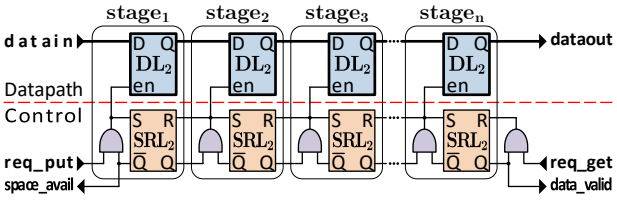

Fig. 3. Asynchronous Symmetric Persistent Pulse Protocol (asP*) pipeline.

B. The FIFO Rings

Figure 4 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 includes a data store unit. 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 5. 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.

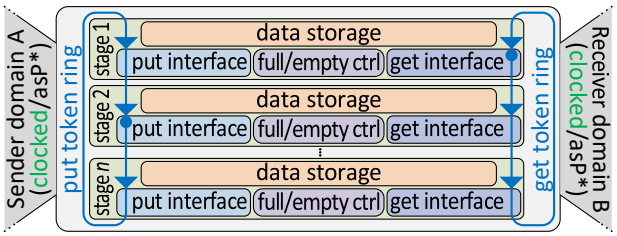

Fig. 4. Architecture overview.

C. Full-Empty Control

Each FIFO stage has one full-empty control block. Figure 6 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 flops f_{ap} and f_{sg} encode the state of the FIFO stage: if the outputs of the 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 fullcycle 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 will not overflow or underflow.

The configurable version of the controller is shown in Figure 6 (bottom). In this version all four mix-and-match combinations are merged and the configuration fields are using to select the `write/read` signals for the clockless configuration interface, or the `clk_put/clk_get` for the clocked configuration. The numbers of flip-flops in the timing-domain crossing synchronizer are configurable, or are completely bypassed if we cross to a clockless domain.

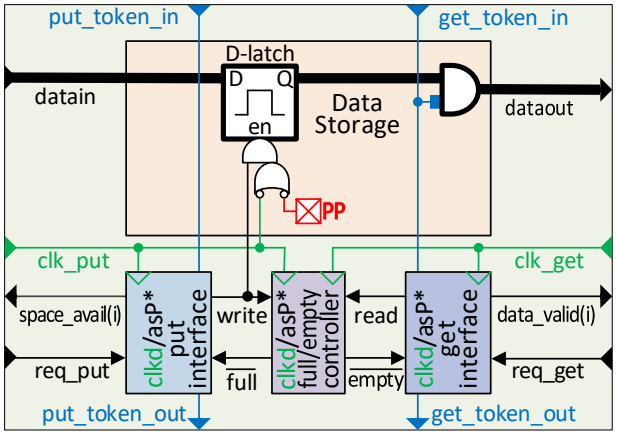

Fig. 5. A single FIFO stage for all mixed-timing combinations.

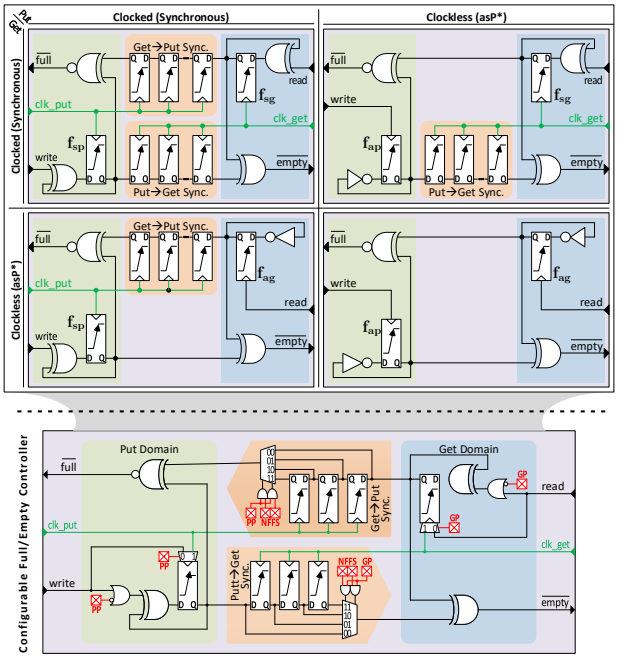

Fig. 6. (top) Full/empty controller for all mixed-timing combinations. (bottom) A reconfigurable version of the full/empty controller.

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 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 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.

Having one synchronizer in each FIFO stage in some situations could incur a prohibitively large area overhead. We compared the area of our design with one using Gray-code counters. Noting that the area of our control circuit is dominated by the area for flip-flops, we compared the two designs based on flip-flop count. For a 16-stage, 32-bit wide FIFO, we estimate that our control circuit is 45% larger than a Gray-code based FIFO, and for a 32-stage FIFO, our control circuit is 54% larger. If the FIFO uses a latch array for storage, this translates to a total area overhead of 16% and 19% respectively. On the other hand, the one-hot encoding provided by the tokens and the single bit-comparison implied by the per-stage synchronizer avoids decode and multi-bit compare operations in the critical path. This allows our FIFO to run at higher speeds than a traditional pointer-based synchronizing FIFO. The higher throughput and greater modularity of our design make these modest overheads acceptable for many designs.

D. The FIFO Interface

The FIFO's top-level signals and stage connectivity are shown Figure 7. 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 stage `space_avail(i)` signals 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 `space_avail(i)` clocked signals from all stages are OR'ed. At any time, one stage at most will raise its `space_avail(i)` signal. A clockless receiver will constantly check `data_valid` signal to examine if the FIDO stores data that can be read. The receiver will reply by raising the signal `req_get`, telling the FIFO the current data has been read and it should move to the next stage. Similarly, on the clocked receiver side, the FIFO tells the receiver that it has valid data at its `dataout` output by raising signal `data_valid`, which is the OR'ed `data_valid(i)` clocked signals.

The FIFO interface units (within each stage) are shown in Figure 8. The get and put rings are part of the get and get 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 indicate the current active stage for get and put interfaces. In the clocked interfaces, `write/read` signals are asserted if the current stage is active, and there is a put/get request, respectively. `space_avail(i)` and `datav_valid(i)` (clocked) 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 the `write/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-flip, and the f_{ap}/f_{ag} flip-flops in the full-empty control, respectively, thus,

$$\text{minPulseWidth}(\text{req\_put}_{\text{LO}}) \geq \text{minPulseWidth}(\text{DFF.CLK}_{\text{HI}}),$$
$$\text{minPulseWidth}(\text{req\_get}_{\text{LO}}) \geq \text{minPulseWidth}(\text{DFF.CLK}_{\text{HI}}).$$

All flip-flops are triggered simultaneously to pass on the put and get tokens. 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 flip-flops' hold time constraints are met. Standard timing tools are used to check and fix these timing constraints as described in Section V.

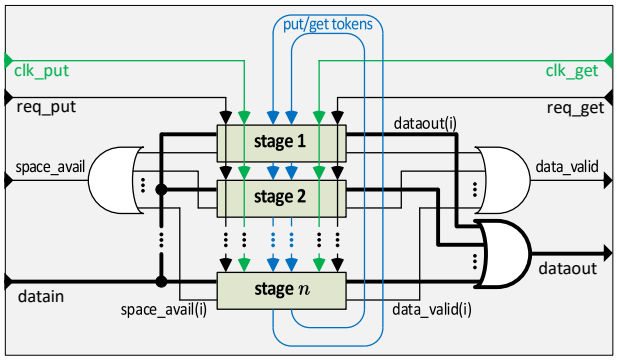

Fig. 7. The FIFO structure for all mixed-timing combinations.

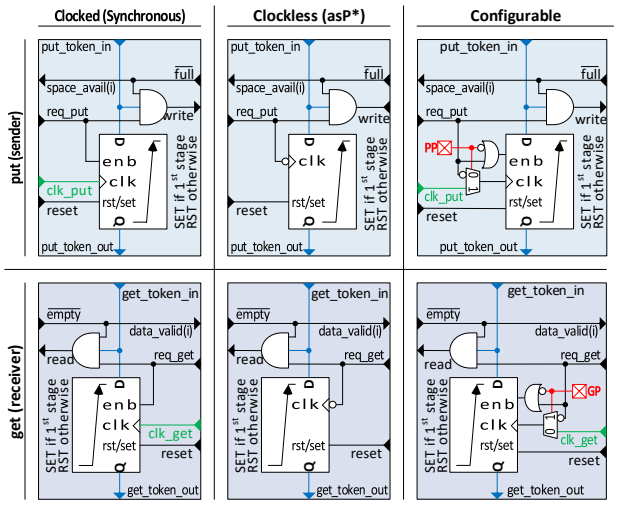

Fig. 8. Interface components.

E. Data Store

To reduce storage area, latches are used to store data instead of registers as described in Figure 5. Each FIFO stage has a latch array for the data; the latch enable is controlled by the stage's `write` signal. Incoming data is propagated 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 passgates, tri-state buffers, or multiplexers. For example, the design described in Figure 7 uses gate-based multiplexers for ease of implementation with a standard CAD 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. If the put protocol is clockless (`PP` field is low), the `clk_put` will be masked, leaving the write signal only to enable the data latch.

IV. Throughput and Latency Analysis

A. Minimum Latency Through the Clockless FIFO

The latency of asP* FIFO stage is measured from a put request is issued by the sender until this stage becomes empty again and stage `full` falls. The chain of events is shown in Figure 9 and is as follows (with the delay in brackets):

1. `req_put` goes high and propagates to all stages [number of gates depends on FIFO depth]
2. `write` goes high in put interface [delay of a 3-input AND gate]
3. FF toggles in full-empty control [2→1 mux + `clk`→`q` delay]
4. Put→get synchronizer is bypassed [delay of a $4 \rightarrow 1$ mux]
5. stage $\overline{\texttt{empty}}$ rises [delay of an XOR gate]
6. `data_valid` OR-tree fan-in [gates depends on FIFO depth]
7. get data rises [delay in receiver]
8. `read` signal rises in get interface (assuming `req_get` is already asserted) [delay of 3-input AND gate]
9. FF toggles in full-empty control [2→1 mux + `clk`→`q` delay]
10. Put→get synchronizer is bypassed [delay of a $4 \rightarrow 1$ mux]
11. stage $\overline{\texttt{full}}$ rises [delay of XOR gate]

B. Minimum Latency Through the Clockled FIFO

The latency through the clocked FIFO is defined as the time from the first write clock edge after a put request until the read clock edge that allows the receiver to retrieve the data. The latency is shortest when the FIFO is empty. The latency path goes from the write clock domain to the read clock domain, so the latency depends on the relationship between the write and the read clock. Figure 10 shows the timing diagram. Counting from the first write clock edge after the put request rises, the latency consists of one write clock cycle, the phase difference, the synchronization latency (in read clock cycles), and one read clock cycle.

C. Throughput of the Clockless FIFO

The throughput of asP* FIFO is defined as the number of data items that the receiver can read out in a given time. Because the clockless FIFO does not need synchronization, as soon as there is data in the FIFO, it can be read out. Consider the case where the put interface is slower than the get interface. Even if the put interface writes data to the FIFO as fast as it can, the get interface will always read it out faster than the next item can be written. At any given time there will be at most one item in the FIFO. The throughput in this situation is determined by the slower put interface. Consider the opposite case where the get interface is slower than the put interface. If the put interface writes data to the FIFO as fast as possible, the FIFO will eventually fill up. The get interface remains busy reading out data as fast as possible. The throughput is now determined by the slower get interface. Assuming the FIFO is written and read as fast as possible, the only situation where its steady state is neither full nor empty is when both put and get interfaces are equally fast. In that case, the FIFO can have any number of data items in a steady state. As long as there is at least one data item, the get interface can read out data items as fast as possible and the throughput is unchanged. In summary, if the FIFO has more than one data item in its steady state, then the get interface is at most as fast as the put interface and the throughput is determined by the get interface. Thus, the throughput is unchanged if there is at least one item in the FIFO. Consequently, having more than one item in the FIFO increases the latency, but does not affect the throughput.

D. Throughput of Clocked FIFO

The throughput of the clocked FIFO depends on the FIFO size. To achieve maximum throughput, the FIFO must have sufficient capacity to cover the synchronization latencies in the full-empty control blocks. In general, a FIFO with equal read and write clock frequencies and a n-FF synchronizer needs $2n + 2$ stages: $2n$ to hide the synchronizer latencies and 2 additional stages to cover the maximum delay before the synchronizers sample a state change. However, if the FIFO has enough stages to cover the synchronization overhead, the throughput is limited by the critical paths from one stage to the next within the put and get interfaces. These paths are longer for larger FIFOs because of the larger fan-outs. Therefore, given a clock ratio and synchronization latency, there is a FIFO size that maximizes the throughput.

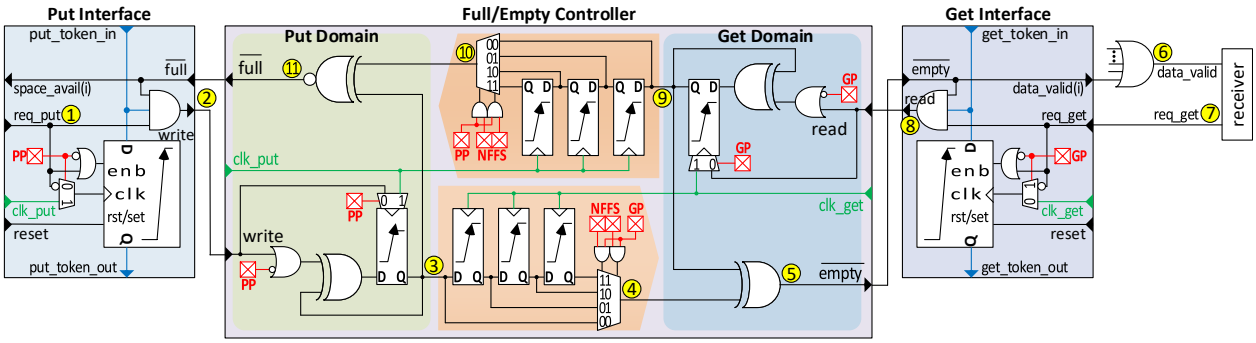

Fig. 9. Latency through the clockless FIFO.

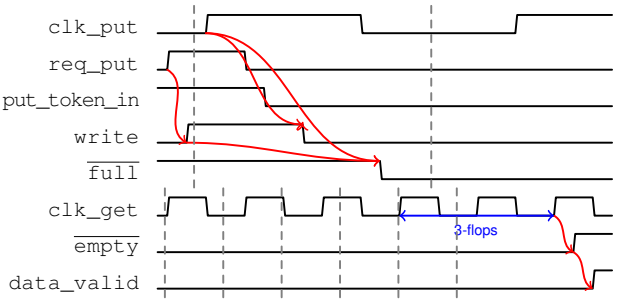

Fig. 10. Latency through the clockled FIFO.

V. FIFO Implementation

To illustrate the suitability of our FIFO design for inclusion in reconfigurable devices, we have implemented the FIFO using a standard ASIC design tools and standard cells starting with a Verilog description and finishing with fully placed and routed gates. Details and results of this process are described in the following subsections.

A. Platform Settings

In order to verify, simulate, and synthesize the suggested approach with various parameters in batch, we wrote a fully parameterized Verilog modules, together with a run-in-batch flow manager that runs Synopsys synthesis tools for each design instance. The Verilog modules and the flow manager are available online [7]. The design package is composed of a fully configurable 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.

Our design framework consists of a run-in-batch manager that 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 flow targets TSMC 65nm technology process and stardard-cell library. The design stages are implemented using Synopsys tools and ordered as follows. (1) Logic synthesis using Synopsys Design Compiler. (2) The cell placement, cell sizing, repeater insertion, and detailed wire routing was performed using using Synopsys IC Compiler; This include delay and parasitic extraction using Synopsys StarRC. (3) Static timing analysis using Synopsys PrimeTime. (4) Functional Gate-level-Simulation was done using Synopsys VCS MX/VCS MXi. with the fully annotated post-layout netlist 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 to estimate both dynamic and leakage power. 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.

B. Experimental Results

The performance of each FIFO configuration is shown in Table III. For the clocked FIFOs we show the maximum clock rate at which that the circuits can operate, while for the asynchronous FIFOs this is the fastest rate at which requests can be serviced. The data in Table III shows that the throughput of the FIFO scales relatively well as the number of stages is increased from 8 to 64. Both clocked and asynchronous FIFOs are able to run at over 1.2 GHz with 64 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 III. 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 5 cycles.

The power measurements are shown in Table III. 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.

Resources consumption is listed in Table IV, and shows that our design consumes $10n$ flip-flops and $32n$ storage latches, where n is the number of stages. This is inline with our design that consumes 10 flip-flops and 32 storage latches (for 32-bit data) per FIFO stage.

Overhead of Distributed Synchronization As noted earlier, our FIFO trades off re-usability and modularity for more area. Unlike a traditional pointer-based FIFO, where the pointer size grows logarithmically with the number of FIFO stages, our FIFO has a distributed synchronization scheme, where every stage has a synchronizers, and grows linearly with the number of stages. As a first-order estimate of the overhead incurred by our FIFO, we counted the number of FFs, the total area, throughput, power consumption, and energy per bit of our FIFO and a traditional gray-code pointer-based

TABLE II
SYNCHRONOUS FIFO AREA COMPARISON

stages	Pointer-based FIFOs					Our synchronous FIFO					Diff. [+%]			
	#FFs	Area [μm^2]	Throughput [Gtransfer/s]	Power [μW]	Energy [fJ/bit]	#FFs	Area [μm^2]	Throughput [Gtransfer/s]	Power [μW]	Energy [fJ/bit]	Area	Throughput	Power	Energy
8	24	4096	1.03	102.64	3.12	80	4582	1.79	118.80	2.31	11.87	73.79	15.75	-33.34
16	32	6166	0.89	122.33	4.30	160	7237	1.58	156.43	3.22	17.37	77.53	28.21	-27.68
32	40	12336	0.72	168.38	7.31	320	15286	1.32	239.52	5.43	23.92	83.34	42.27	-22.30
64	48	20568	0.59	226.60	12.01	640	30745	1.21	391.37	10.11	49.48	105.09	72.74	-15.83

TABLE III
PERFORMANCE AND POWER CONSUMPTION (WITH 3-FF SYNCHRONIZERS)

FIFO Configuration			Throughput	Latency	Power	Energy
Put	Get	Stages	[Gtransfers/s]	[ns]	[μW]	[fJ/bit]
clkd	clkd	8	1.79	4.15	118.80	2.31
clkd	clkd	16	1.58	4.39	156.43	3.22
clkd	clkd	32	1.32	4.90	239.52	5.43
clkd	clkd	64	1.21	5.63	391.37	10.11
asP*	clkd	8	1.63	3.87	109.67	2.11
asP*	clkd	16	1.58	4.16	219.12	4.34
asP*	clkd	32	1.37	4.50	351.44	8.02
asP*	clkd	64	1.19	4.98	516.40	13.57
clkd	asP*	8	1.63	3.86	107.23	2.06
clkd	asP*	16	1.60	3.98	173.74	3.40
clkd	asP*	32	1.48	4.42	361.80	7.64
clkd	asP*	64	1.22	5.92	575.66	14.75
asP*	asP*	8	1.75	0.49	102.65	1.84
asP*	asP*	16	1.66	0.58	246.50	4.65
asP*	asP*	32	1.50	0.65	376.21	7.84
asP*	asP*	64	1.34	0.77	616.43	14.38

TABLE IV
RESOURCES CONSUMPTION OF MULTIPLE FIFO INSTANCES (32-BIT DATA)

Stages	Total Area [μm^2]	#flops	#latches	Datapath [%]
8	4581.71	80	256	46.74
16	7236.20	160	512	52.30
32	15285.34	320	1024	55.15
64	30744.82	640	2048	56.59

FIFO (Table II). As expected, compared to a gray-code pointer-based FIFO our FIFO can consume up 50% more silicon area for the largest configuration, on the other hand, our FIFO doubles the throughput for the same configuration. Energy efficiency is improved for at least 15% for all configurations.

VI. CONCLUSIONS AND FUTURE WORK

This paper presents configurable high-performance synthesizable interfaces for crossing between asynchronous and synchronous timing domain. The interface protocol can be configured as synchronous or asynchronous (asP*) protocol. In addition, the number of FIFO slots and the number of synchronization flip-flops are also configurable. The configurability, performance, and vast applications of our FIFO make it suitable for hard-coded integration in reconfigurable devices.

As a future work, we are planning to support SRAM-based storage [16] instead of latch-based storage to increase area efficiency. Furthermore, we are planning to investigate searchable FIFO structures known as Content-Addressable FIFOs [17], enabling configurability via hierarchical content-addressable memories [18]–[20]. Finally, it is advantageous to support the two-phase asynchronous protocol in our configurable FIFO due to its power and speed efficiency.

REFERENCES

[1] A. Abdelhadi, R. Ginosar, A. Kolodny, and E. G. Friedman, "Timing–driven variation–aware synthesis of hybrid mesh/tree clock distribution networks," *Integration*, vol. 46, no. 4, pp. 382–391, 2013.

[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 Systems*, vol. 30, no. 9, pp. 1387–1399, Sep. 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] 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.

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

[8] 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.

[9] *FIFO Intel FPGA IP User Guide: Updated for Intel Quartus Prime Design Suite 18.06*, Intel Corp., Nov. 2019 (accessed Sept. 10, 2020).

[10] *Vivado Design Suite LogiCORE IP Product Guide (PG057): FIFO Generator v13.2*, Xilinx Inc., Oct. 2017 (accessed Sept. 10, 2020). [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v13_2/pg057-fifo-generator.pdf

[11] T. Chelcea and S. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Trans. on Very Large Scale Integration Syst. (TVLSI)*, 2004.

[12] 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)*, 2017.

[13] 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.

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

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

[16] A. M. S. Abdelhadi and G. G. F. Lemieux, "A Multi-ported Memory Compiler Utilizing True Dual-Port BRAMs," in *Proc. of the IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016.

[17] K. Schultz and P. Gulak, "Multicast contention resolution with single-cycle windowing using content addressable FIFO's," *IEEE/ACM Trans. on Networking*, vol. 4, no. 5, pp. 731–742, 1996.

[18] 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 Technology (FPT)*, 2014.

[19] A. M. S. Abdelhadi and G. G. F. Lemieux, "Modular SRAM-Based Binary Content-Addressable Memories," in *Proc. of the IEEE Annu. Int. Symp. on Field-Program. Custom Comput. Mach. (FCCM)*, 2015.

[20] 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)*, 2018.