Two-Phase Asynchronous to Synchronous Interfaces for an Open-Source Bundled-Data Flow

Ameer Abdelhadi*, Dake Chen†, Huimei Cheng†, Gourav Datta†, Yang Zhang†, Peter Beerel†, Mark Greenstreet‡

*Electrical & Electronic Engineering, Imperial College London, a.abdelhadi@imperial.ac.uk
†Ming Hsieh Electrical and Computer Engineering, University of Southern California, {dakechen,huimeich,gdatta,zhan808,pabeerel}@usc.edu
‡Dept. of Computer Science, University of British Columbia, mrg@cs.ubc.ca

***Abstract*—This paper proposes a family of modular interfaces for crossing between asynchronous and synchronous timing domains. Motivated by asynchronous NoC and desynchronized pipelines, these converters support two-phase, bundled data communication. They provide high-throughput communication and support an early-request protocol that can hide most of the synchronization latency. Our designs are fully synthesizable using widely available standard cell libraries and a standard ASIC design flow. The converters have been integrated into an open-source bundled-data desynchronization flow and evaluated in isolation as well in the context of an encryption engine supporting three block ciphers.**

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.

As an example, Figure 1 depicts a simplified view of a modern, multi-core CPU that has a crytpo-accelerator block. 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 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 [1]–[5], the twelve cycle synchronization penalty of the all-synchronous example design can be alleviated to half that, *i.e.* six cycles. Further reductions in the synchronization penalty are possible by using an *early-request* protocol when the maximum delay in a module can be bounded.

It may also be advantageous that the CPU or the accelerators be asynchronous. There are several asynchronous design languages that support the design of asynchronous blocks from scratch [6], [7]. Alternatively, desynchronization is an attractive approach which leverages existing synchronous RTL specifications, commercial synthesis tools, and cell libraries to generate an asynchronous bundled-data pipeline [8]. Because the pipeline uses local clocks generated via asynchronous control, the pipeline is resilient to process, voltage, and temperature (PVT) variations [9], [10] and can easily benefit from dynamic voltage scaling. As in the ANoC, a critical aspect of making such systems usable is the integration with low-latency synchronizers. Moreover, because desynchronization techniques yield unconditional pipelines they are particularly well suited to take advantage of early-request protocols.

This paper thus focuses on interfaces between synchronous and asynchronous timing domains that support early request protocols. Our interfaces assume two-phase, bundled data protocols 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 a two-phase protocol, each data transfer requires a single round-trip between the sender and receiver: first, data and a request are sent from the sender to the receiver; in response, the receiver sends an acknowledgement back to the sender. The throughput of the network is limited by this round-trip time. If a four-phase protocol is used, two such round-trips are required for each data transfer, achieving roughly half the throughput of a two-phase design. We describe how the early-request protocol can be incorporated into our desynchronization process and evaluate it on an encryption accelerator that

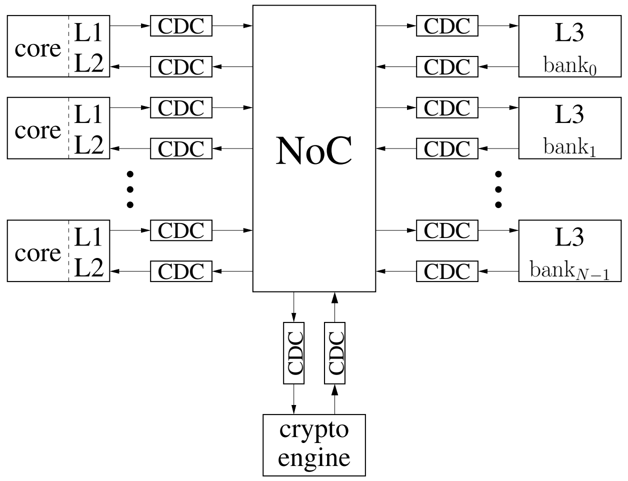

Fig. 1: CDC example: A multi-core CPU with a crypto accelerator

supports three block ciphers, Triple DES, Present, and Hight.

The contributions of this paper are as follows.

- We present a family of timing-domain crossing FIFOs that support any combination of synchronous and asynchronous interfaces, *i.e.* A2A, A2S, S2A, and S2S.
- The designs are highly parameterized and synthesizable using standard, commercial cell libraries and design flows. We provide open-source Verilog including the FIFOs themselves and their test benches.[1]
- Our designs support two-phase, bundled-data with early-request protocols. We present simulation results showing how the use of early request can significantly reduce the latency of A2S and S2S interfaces.
- We evaluated our FIFOs in isolation as well in the context of an encryption engine supporting three block ciphers.

By providing a synthesizable library of high-throughput, low-latency time-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.

The remainder of this paper is organized as follows. Section II reviews related work in synchronizers and desynchronization. Section III presents our proposed synchronizing FIFOs. Section IV focuses on our proposed early-request protocols and their implementation. Section V presents our experimental results followed by a summary in Section VI.

II. 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 more attractive solution for decoupling and transferring data between different domains because they offer high throughput and simple flow control. Synchronizing FIFOs are largely classified by the design of the interface control logic, the data storage, and the synchronization mechanisms between the interfaces. FIFOs based on Gray code counters [11], [12] are the most common synchronizing FIFOs. On a transition between two successive numbers, exactly one bit of a Gray code counter makes a transition which is very useful for synchronization. The disadvantage of Gray codes is the difficulty of comparing two Gray code values to determine which is greater. This tends to be a slow operation that limits FIFO performance. Like several other designs [13]–[16], 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 the first to support two-phase asynchronous protocols. 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.

A novel feature of our design is the use of an early-request protocol to reduce the latency penalty of synchronization when transfering data to a synchronous timing domain. The idea is that the sender can make an "early request" before the data to be sent is available, as long as there is bound on when that data will be available and the corresponding "real" request will be made. We are aware of prior work exploring speculative synchronization [17] that uses speculation, error-detection, and retry to reduce the average synchronizer latency. This is similar to the speculation techniques of Razor [18]. In contrast with these methods, our early-request protocol does not require an error-detection, roll-back, and retry mechanisms. Instead, we use static timing analysis to generate a safe, early-request. This approach is well-suited for desynchronization based designs where the data-flow is readily identified.

Related desynchronizing efforts targeted both quasi-delay-insensitive dual-rail circuits as well as bundled-data design that use standard-cell single-rail logic [8], [19]–[22]. While most of these approaches propose interfacing with synchronous modules, they do not focus on the integration of the required clock-domain-crossing logic. While applicable to all of these techniques, our designs directly interface with desynchronization approaches that target the two-phase bundled-data protocol.

III. The FIFOs

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, or anything in between. From the synchronous designer's perspective, the asynchronous module(s) communicate through latency-insensitive interfaces; and no special considerations

[1]For the purposes of evaluation, we will provide a tar-ball to the PC chairs upon request. After the blind review process is complete, we will post the source code on git-hub.

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 a two-phase, bundled data protocol. The asynchronous-to-synchronous (A2C) converter is the inverse: in 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 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.

A. A2S – The Asynchronous-to-Synchronous Interface

We start with the A2S interface, as the others can be understood as fairly simple modifications of this design. The A2S module comprises a two-phase, bundled data, asynchronous read interface and a positive-edge triggered, synchronous write-interface. A write is performed by providing data at *dIn* and toggling *wReq*. The A2S interface toggles *wAck* to indicate the transfer is complete and a new transfer can be initiated. Data is output on *dOut* and the dValid signal indicates that the data has not yet been consumed by a read-request, *i.e.* asserting *rReq*. The *dOut*, dValid outputs are synchronous to the receiver's clock, *rClk* as is the *rReq* input. The interface supports a throughput of one data word per clock-cycle. The receiver can continuously assert *rReq*, and the interface will assert *dValid* on cycles where data is available on *dOut*. If *rReq* is asserted on a cycle when no data is available, the request is ignored.

Figure 2 (top) shows the control logic. From the synchronous designer's perspective, the interface implements a simple, latency insensitive protocol. When data is available from the FIFO, *dValid* will be high. The synchronous domain takes the data by asserting *rReq*. The data will be output on the next rising edge of the clock, and *dValid* is updated on the same cycle. If there is data in the FIFO, a new value can be removed on each cycle of *rClk*. The outputs of the A2S interface are properly synchronized – thus preserving the functionality of a latency-insensitive design.

The asynchronous write-interface and synchronous read interface each has a Johnson counter. When $wVac_i = rVac_i$, stage i is empty; conversely, when $wVac\_i \neq rVac_i i$, stage i is full – *data_latch[i]* holds data that has been written by the sender and is waiting to be consumed by the receiver. For each Johnson counter, there is exactly one flip-flop for which Q $\neq$ D; this is the flip-flop that will transition on the next write or read event, and is a pointer to the latch to be written or read.

Consider an initial condition where all of the flip-flops in the two Johnson counters are in the Q $=0$ state – the FIFO is empty. On the other hand, $f_i = 0$ for every stage, thus $dValid = 0$. Furthermore, assume $wReq = wAck = 0$, and $rClk = 0$. Because $wReq = wAck = 0$, the sender is allowed to insert a new value into the FIFO by applying data at *dIn* and transitioning *wReq* to 1. The rising edge of *wReq* propagates to w_0 and triggers $wVac_0$ and *wAck_flip* to transition to 1. The transition of $wVac_0$ to 1 causes e_0 to fall, and the data is stored in *data_latch[0]*. The rising edge of *wAck_flip* triggers the flip-flop that drive *wAck* and acknowledges the data transfer. The falling edge of e_0 causes w_0 to return to 0 which then causes *wAck_flip* to return to 0 as well. Now, the flip-flop driving $wVac_1$ is the active stage. The next request is signaled by a falling edge of *wReq* which causes an equivalent sequence of transitions on w_1, $wVac_1$, e_1, *wAck_flip*, and *wAck*, and stores the next data value into *data_latch[1]*. In this manner, values can be loaded into the FIFO until a state is reached where stage i is the next to write but $wVac\_i \neq rVac_i i$. This indicates that the stage is full, and a transition on *wReq* will not be acknowledged until $rVac_i$ transitions to indicate that the stage is empty and a new value can be stored in *data_latch[i]*.

Read operations are similar; for the A2S, the read logic is synchronous. Each $wVac_i$ signal is passed through a synchronizer so it can be safely used in the synchronous domain and compared with $rVac_i$. If the synchronized version of $wVac_i$ is not equal to $rVac_i$, then the *data_latch[i]* is full and the data can be read. The signal $rStg_i$ indicates the currently active stage, and controls the multiplexor that selects a data-latch to output *dOut_prev*. The output is registered to prevent fall-through glitches.

This simplicity of the control circuitry makes it modular. The FIFO capacity, word width, and synchronizer depth can all be changed independently. Likewise, changing configurations to produce S2A or S2S designs is straightforward as described below. The two-phase protocol appears to simplify the implementation of the synchronizers. For example, the design described in [16] uses an asymmetric synchronizer where all stages of the synchronizer are reset by the receiver when a data value is removed. With the two-phase protocol, successive requests are indicated by a *transition* of the synchronizer output, and no reset is required.

B. S2A – the Synchronous-to-Asynchronous Interface

The S2A interface is shown in the left side of Figure 4. It is the complement of the A2S. The S2A provides a positive-edge triggered, synchronous write-interface, and a two-phase, bundled data, asynchronous read interface. The synchronous writer provides data at *dIn* and asserts *wReq* prior to a rising edge of its clock, *wClk*. If there is space available as indicated by *SpaceAv*, the write will be performed; otherwise it is silently ignored. The *SpaceAv* output is synchronous to *wClk*. A write can be performed on every cycle of *rClk* by continuously asserting *rReq* and confirming that there was space in the FIFO and that the write was performed by checking the value of *SpaceAv*.

The implementation of the S2A is very similar to that of the A2S. The main difference is that there is no synchronization on the request path – data can be transferred directly from

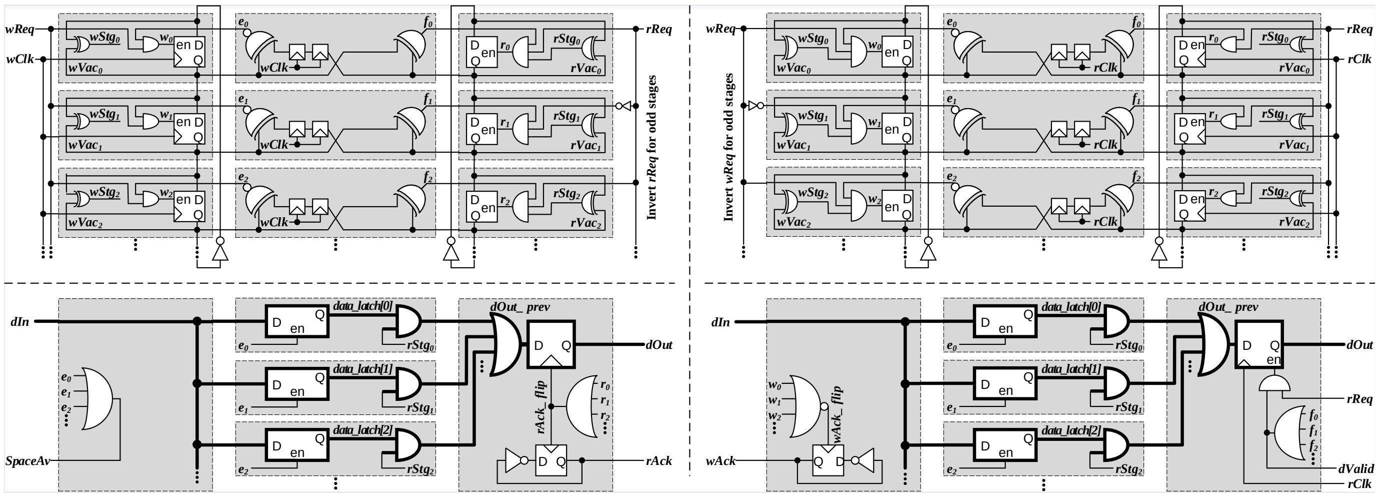
Fig. 2: Mixed-timing FIFO, (top) control logicc (bottom) data path, (left) A2S FIFO, and (right) S2A FIFO.

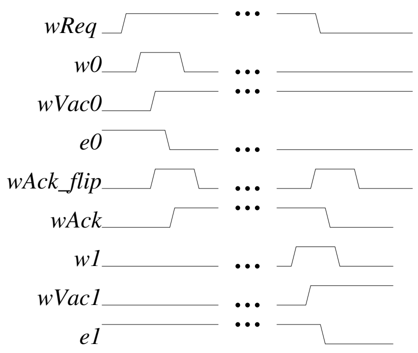

Fig. 3: A2S – Writing Data

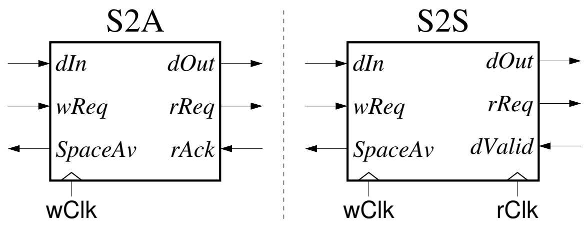

Fig. 4: The S2A and S2S interfaces

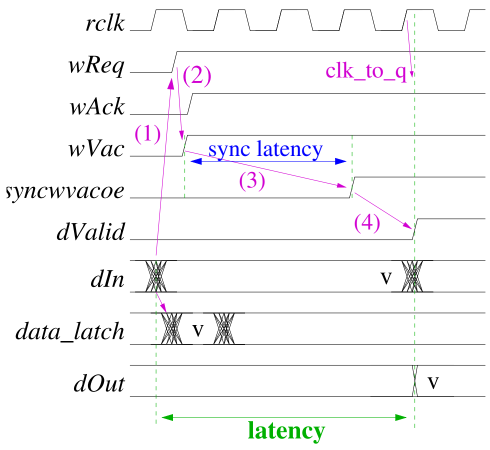

Fig. 5: Latency of the A2S interface without early request

the synchronous domain to the asynchronous block without synchronization, and thus with no latency penalty. The synchronizer appears instead on the acknowledgement path. This latency can be "hidden" by using a FIFO of sufficient capacity. A minimum capacity of two plus the number of synchronizer flip-flop stages is required to maintain one transfer per clock cycle. In practice, the FIFO should be sized slightly larger than this to include the req-to-ack delay of the asynchronous block in the timing budget.

C. S2S – the Synchronous-to-Synchronous Interface

While this paper focuses on A2S and S2A, the modularity of our design supports S2S interfaces, allowing a designer to use one, parameterized design for all time-domain-crossing interfaces. The S2S interface is shown in the right side of Figure 4. The implementation consists of the synchronous write interface of the S2A and the synchronous read interface of the A2S. This clock-domain crossing interface has a throughput of one data transfer for each cycle of the slower clock. To do so, the FIFO capacity must be at least four plus twice the number of flip-flop stages in each synchronizer.

IV. Early Request

Figure 5 shows how data propagates through the A2S interface. We measure the *latency* of the interface as the time from when *dIn* has settled to a value, v, until that value is available to the receiver at *dOut*. The data from the sender, *dIn* must have its stable value, v, before the transition on *wReq* requesting the transfer (arrow 1). The transitionon *wReq* causes the *wVac* signal for the active stage to transition (2), which propates through the synchronizer (3), to assert *dValid* and output v on *dOut*. In the latency critical case, the receiver is waiting for *dValid*, and we consider the simplest version where the FIFO was empty prior to inserting v. In this case, *data_latch* was transparent from the outset, and the data was waiting at the output of the latch during the synchronization of the control signal. There are no synchronizers for the data itself, but *dValid* is not signaled until the synchronization in the control path is complete. A key observation is that the data is not needed until the synchronization is complete. This motivates the *early request* designs described here.

Often, we can determine when data will be output by an asynchronous block before the data is actually available. For example, our open-source desynchronization flow excludes arbitration and conditional branching, and bounds on the arrival time of a data value at the output of a pipeline can be determined by earlier stages[2]. In this case, the asynchronous block can generate an *early request*, *wReqE*. An early request is a guarantee that the actual request will be asserted within a specified time bound – we describe the details of this bound shortly.

Figure 6 shows the control-path for A2S interface with early request; the data path is the same as shown in Figure 2 (bottom, left). The interface has two Johnson counters in the write interface. The "early-counter" shown in red handles the handshake with *wReqE* and *wAckE*. The $wVacE_i$ signals generated by this counter are synchronized by the receiver to determine when data is available. The *wReq* signal indicates the actual arrival of data. This is used to generate the e_i signals that control when the data latches are transparent. In particular, *data_latch[i]* goes opaque when e_i falls, indicating that the anticipated data has arrived.

Figure 7 shows how data propagates through the A2S interface with early request. In this design, a transition on *wReqE* triggers an transition on $wVacE_i$ (arrow 1), where i is the cuurently active stage. The transition on $wVac0_i$ then propagates through the synchronizer (arrow 2), and the synchronized result triggers the assertion of *dValid* and the update of *dOut* (arrow 3).

The lower part of Figure 7 shows the data-path. The data at *dIn* propagates to *data_latch[i]* (arrow 4). If the latch is empty, *dIn* propagates immediately (as shown in the figure); otherwise, it waits for transition on $rVac_i$, delaying *wAck* as well. The output of *data_latch[i]* is transfered to *dOut* (arrow 5) on the edge of *rClk* determined by the control path as described above. The figure also shows the bundling constraint that *dIn* must settle before the transition on *wReq* (arrow 6); the interface responds to *wReq* with a transition on *wAck* (arrow 7); and that the sender must maintain stable data on *dIn* until after receiving the acknowledgment on *wAck* (arrow 8).

Comparing with Figure 5, we see that the *wReqE* signal takes the role of the *wReq* signal for generating a synchronized control signal in the receiver's domain. However, the *dIn* does not need to settle until shortly before the edge of *rClk* that sets *dValid* to true. In other words, *dIn* can settle roughly the synchronizer delay *after* the *wReqE* event. If the delay of the asynchronous pipeline is sufficiently long and predictable, then most of the synchronizer latency can be hidden. In practice, we expect an asynchronous block to perform a substantial amount of computation – if its function were trivial, a synchronous implementation would be just as good. Furthermore, static-timing analysis tools are now sufficiently accurate to provide reliable bounds on the worst-case delays of paths in the asynchronous block.

[2]If there is back pressure, it comes from the A2S interface itself, and data is guaranteed to arrive if each stage of the asynchronous block has a cycle time that is less than the clock period of the synchronous circuit.

A. Timing Constraints

For correct operation, data must arrive at the D input of the *dOut* flip-flop early enough to satisfy the set-up requirement on the cycle that the synchronized *wReqE* propagates to the enable input. Thus, we need a lower bound for the delay from *wReqE* to the edge of *rClk* that uses the resulting assertions of the en input of the *dOut* flip-flop. We also need an upper bound for the delay from *dIn* to the D input of the same flip-flop.

Let N_{sync} denote the number of flip-flops in the synchronizer and P denote the period of *rClk*. If the input to the synchronizer changes just before a rising edge of *rClk*, then it will propagate to the output of the synchronizer in time $(N_{\text{sync}} - 1)P$ plus some adjustments for the hold-time of the first flip-flop and the clock-to-Q time of the last one. The path from *wReqE* to en consists of a combinational logic path including the hold time for the first flip-flop of the synchronizer plus $(N_{\text{sync}} - 1)P$ for the synchronizer, plus P because en is used one clock cycle after the synchronizer output changes. Let $\delta_{\text{ctrl0,min}}$ be the minimum delay of this combinational logic path. Let $\delta_{\text{ctrl,min}}$ denote the minimum delay from a transition on *wReqE* to an

$$\delta_{\text{ctrl,min}} \quad = \quad \delta_{\text{ctrl0,min}} + N_{\text{sync}}P \tag{1}$$

The upper bound for the data path is straightforward. This is the D-to-Q delay of the transparent latches plus the delay of the data multiplexor (shown as 2-input AND-gates and a wide OR-gate) plus the set-up time for the *dOut* flip-flop. Let $\delta_{\text{data,max}}$ be the maximum delay for this path. Let κ denote the "kiting delay", i.e., the amount of time by which *wReq* can anticipate the arrival of data at *dIn*. The key constraint is

$$\begin{aligned} \kappa &= \delta_{\text{ctrl,min}} - \delta_{\text{data,max}} \\ &= N_{\text{sync}}P + \delta_{\text{ctrl0,min}} - \delta_{\text{data,max}} \end{aligned} \tag{2}$$

Note that κ grows as $N_{\text{sync}}P$. This means that by using early-request, we can hide most of the synchronizer latency with the operation time of the asynchronous module. The synchronization overhead remains non-zero because of the static timing-analysis margins (i.e. using max-delays for δ_{data} and min-delays for δ_{ctrl0}). The synchronizer adds an additional latency between 0 and P because the incoming data must wait until the next edge of *rClk* to be transfered to the receiver's clock domain.

B. Simply vs. Aggressively Early

In practice, we found that there are two approaches to exploiting early requests. The first approach assumes that $\kappa < P$ – this means that the *wReqE* transition for the k^{th} data transfer occurs after the *wReq* event for transfer $k - 1$. In this case κ just the time from a transition on *wReqE* to a

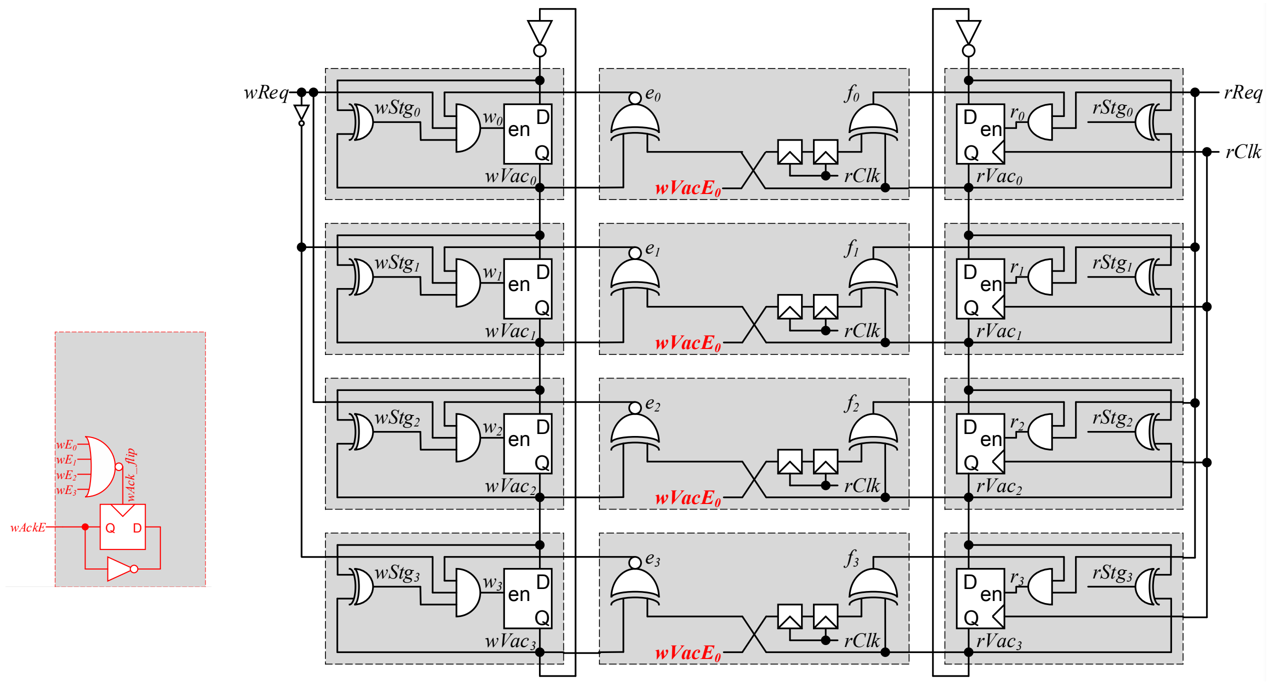

Fig. 6: Control of A2S with early request (4-stages example)

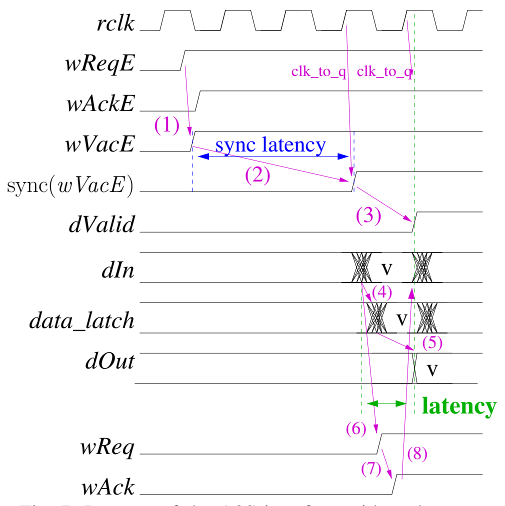

Fig. 7: Latency of the A2S interface with early request

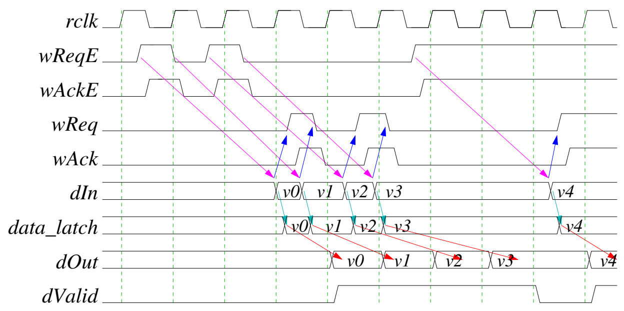

Fig. 8: Operation of the A2S interface with aggressive early request

transition on *wReqE*. Generating the SDC timing constraints is straightforward and can be easily generated automatically. The performance cost is that this simple form of early-request can hide at most one clock period of synchronizer latency. This is acceptable with a two, flip-flop synchronize where that's the maximum latency that can be removed. For higher clock frequency designs, synchronizers with three or four flip-flops are needed to achieve acceptable failure rates. In this case, the more aggressive use of early-request described below can offer better performance.

A more aggressive use of early-request allows $\kappa > P$ a transition on *wReqE* to be followed by further *wReqE* events before the *wReq* event occurs. The design shown in Figure 6 supports this aggressive use of early request, and this is the logic produced by our Verilog code. Figure 8 depicts the operation of this design, again assuming three flip-flop synchronizers. The magenta arrows show the timing-constraint for the early-request: *dIn* must settle within κ time units after the *wReqE* event. The blue arrows show the data bundling constraint: *dIn* must settle before the *wReq*. The red arrow shows the propagation of data values from the data latches to *dOut*. *dOut* changes synchronously on the rising edge of *rClk*, but not necessarily on the *first* rising edge after the data is in the latch. In the example in Figure 8, the asynchronous block produces a burst of four data items, $v0, \ldots, v3$, with a cycle time less than the period of *rClk*, followed by a pause and then a fifth data value, $v4$. The Johnson counters in the FIFO ensure that these values are output in the correct order, and some values (e.g. $v2$ and $v3$) are delayed to a later clock edge.

Checking the timing constraints requires keeping track of which *wReqE* event corresponds to *wReq* event. Currently, we construct these constraints manually. In summary, the aggressive approach offers better performance at a cost of some manual checking of timing constraints.

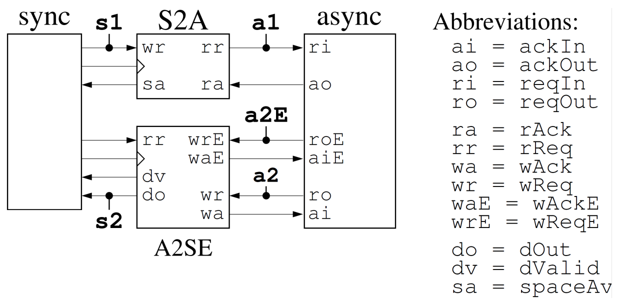

Fig. 9: Measurement set up for evaluating the latency of early requests

C. Early Request and the S2A and S2S interfaces

Although there is no synchronization in the data path of the S2A interface, the early request technique still finds application. The most-common application of a desynchronization flow is to incorporate asynchronous blocks in a synchronous design. In this case, a synchronous block that communicates through a S2A can provide an early request to notify that asynchronous block that a request is forthcoming. The asynchronous block can use this early request to generate its own early request to a downstream A2S, thus lowering the latency at that interface. This approach is particularly attractive when the synchronous can provide an early request a known number of clock cycles before the actual request. This gives the designer a very precise specification of the amount by which the early request leads the actual request.

In a similar manner, when the synchronous interface can provide an early request a known number of clock cycles before the actual request, the S2S interface can exploit this to hide synchronization latency. In both the S2S and S2A case, the clock periods of the send and receiving domains must be specified to verify the timing constraints for the early request.

V. Experimental Results

A. An Open Source IP and Design Flow

In order to verify and evaluate our proposed family of FIFOs, we designed a fully parameterized open-source CDC framework. The framework include a Verilog description of the proposed converters, associated Verilog testbench, a Synopsys design constraint file, tool configuration scripts, and a run-in-batch manager. The Verilog description of the proposed FIFO is parameterized and can be synthesized in isolation or automatically added to desynchronized blocks. The package also includes a run-in-batch manager that allows synthesizing and simulating a number of FIFOs in batch. The testbench will generates input vectors, check the outputs, and compare them against a generic FIFO.

B. Early Request

We set up a test for measuring CDC latency as shows in Figure 9 using 4-stage A2S and S2A synchronizers with a synchronous interface running at 500MhZ. The asynchronous block has an adjustable latency from a1 to a2 of δ_{async}, which we implemented by a programmable delay line. δ_{sync} is the time a token traveling from s1 to a1, then processed by asynchronous block and transferring through A2S interfaces.

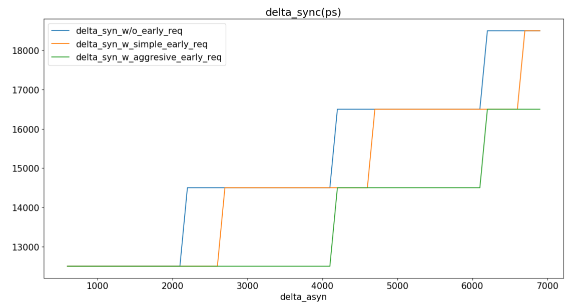

Fig. 10: Comparison of early-request protocols in terms of δ_{sync}

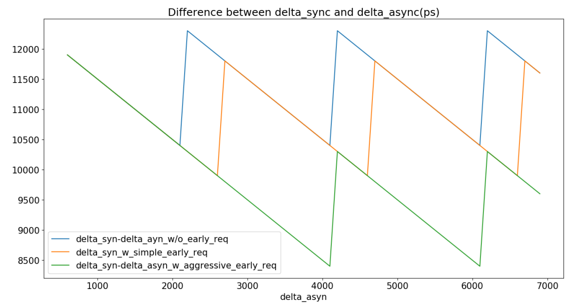

Fig. 11: Comparison of early-request protocols in terms of $\delta_{sync} - \delta_{async}$

Therefore, the difference between δ_{sync} and δ_{async} is the delay stemming from S2A and A2S interfaces. Because of the early request, part of logic of asynchronous block overlaps with some preparing actions in the A2S, as we increase δ_{async}, δ_{sync} will not increase as fast as the counterpart without early request, thereby the difference of δ_{sync} and δ_{async} also drops, which means the latency decreases. This is illustrated in Figures 10 and 11.

C. Area and Max Frequency

Figure 12 and Figure 13 present the area and maximum frequency of S2A and A2S with different sizes obtained from Synopsys's Design Compiler Framework using a 28nm FDSOI cell library. We swept the data width from 8 to 32 bits and the FIFO depth from 4 to 32. The figures show the results for synchronizer depth of 4 to provide reasonable mean-time-before-failure for the obtained high frequencies. The area and frequency trends for other depths are similar. In particular, for all designs, the maximum read-side frequency is from 4-5 GHz with the higher rates for designs with smaller widths and/or depths. The area is proportional to the data width and fifo depth.

D. Integration with a Desynchronization Flow

The CDC framework has been incorporated into an open-source desynchronization flow [23]. The desynchronization flow optionally surrounds the desynchronized block with an A2S and S2As to more easily integrate it into synchronous SoC. An extra output from the delay line to the A2S is tapped to produce the early request, as shown in Figure 14. Both the early and total delays are programmable. We tested the

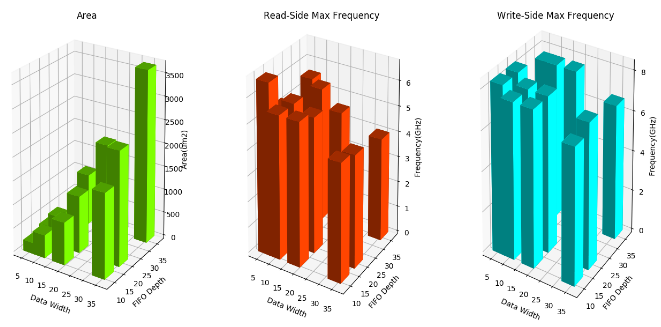

Fig. 12: Area and max frequency of different size S2A modules

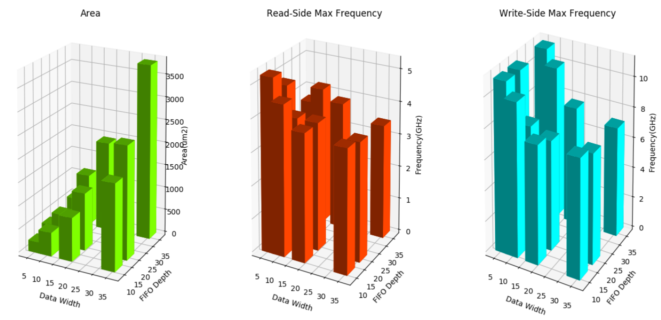

Fig. 13: Area and max frequency of different sized A2S modules

CDC integration on a de-synchronized encryption engine that includes pipelined implementations of three block ciphers, Triple DES, Present, and Hight. The latency improvement is consistent with the simple early request in Figures 10 and 11 with a maximum estimated total latency savings of 5%-10% with a synchronous frequency of 500 MhZ. Our future work includes integrating the aggressive early request which involves tapping the early request from earlier in the pipeline and automatically pipelining the handshaking signals between it and the A2S.

VI. Summary and Conclusions

This paper presents open-source high-performance synthesizable interfaces for crossing between asynchronous and synchronous timing domain. The interfaces support an early-request two-phase bundled-data protocol that can hide most of the synchronization latency. The circuits have integrated into the open-source desynchronization flow and evaluated both in isolation as well in the context of an encryption engine supporting three block ciphers, Triple DES, Present, and Hight.

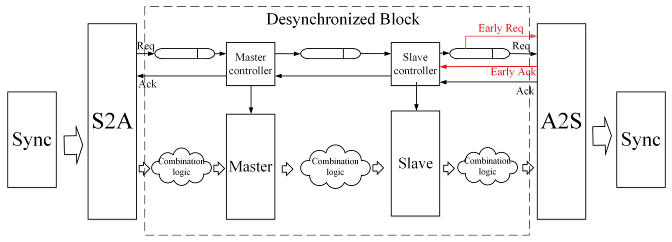

Fig. 14: Integration of early request CDCs into a desynchronization flow

References

[1] D. Lattard, E. Beigne, F. Clermidy, Y. Durand, R. Lemaire, P. Vivet, and F. Berens, "A reconfigurable baseband platform based on an asynchronous network-on-chip," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 223–235, Jan 2008.

[2] D. Gebhardt, J. You, and K. S. Stevens, "Design of an energy-efficient asynchronous noc and its optimization tools for heterogeneous socs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 9, pp. 1387–1399, Sep. 2011.

[3] 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.

[4] R. Dobkin, R. Ginosar, and I. Cidon, "Qnoc asynchronous router with dynamic virtual channel allocation," in *First International Symposium on Networks-on-Chip (NOCS'07)*, May 2007, pp. 218–218.

[5] J. Bainbridge and S. Furber, "Chain: a delay-insensitive chip area interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, Sep. 2002.

[6] A. Bardsley and D. A. Edwards, "The balsa asynchronous circuit synthesis system," 2000.

[7] Tiempo, "White Paper No. 2: Introduction to SystemVerilog asynchronous modeling," 2011.

[8] J. Cortadella, A. Kondratyev, L. Lavagno, and C. Sotiriou, "Desynchronization: Synthesis of asynchronous circuits from synchronous specifications," *IEEE Trans. on CAD*, vol. 25, no. 10, pp. 1904–1921, Oct 2006.

[9] J. Cortadella, M. Lupon, A. Moreno, A. Roca, and S. S. Sapatnekar, "Ring oscillator clocks and margins," in *Proceeding of 2016 Symposium on Asynchronous Circuits and Systems*, May 2016.

[10] Y. Zhang, H. Zha, V. Sahir, H. Cheng, and P. Beerel, "Test margin and yield in bundled data and ring-oscillator based designs," in *23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2017, pp. 85–93.

[11] 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 Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 10, pp. 1125–1134, Oct 2007.

[12] C. Cummings and P. Alfke, "Simulation and synthesis techniques for asynchronous FIFO design with asynchronous pointer comparison," in *SNUG-2002*, 2002.

[13] T. Chelcea and S. M. Nowick, "Robust interfaces for mixed-timing systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 8, pp. 857–873, Aug 2004.

[14] I. M. Panades and A. Greiner, "Bi-Synchronous FIFO for Synchronous Circuit Communication Well Suited for Network-on-Chip in GALS Architectures," in *First International Symposium on Networks-on-Chip (NOCS'07)*, May 2007, pp. 83–94.

[15] T. Ono and M. Greenstreet, "A modular synchronizing FIFO for NoCs," in *2009 3rd ACM/IEEE International Symposium on Networks-on-Chip*, May 2009, pp. 224–233.

[16] A. M. S. Abdelhadi and M. R. Greenstreet, "Interleaved Architectures for High-Throughput Synthesizable Synchronization FIFOs," in *2017 23rd IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, May 2017, pp. 41–48.

[17] G. Tarawneh, A. Yakovlev, and T. Mak, "Eliminating synchronization latency using sequenced latching," *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 2, pp. 408–419, Feb. 2014.

[18] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, Dec 2003, pp. 7–18.

[19] Y. Thonnart, E. Beigne, and P. Vivet, "A pseudo-synchronous implementation flow for WCHB QDI asynchronous circuits," in *ASYNC*, 2012, pp. 73–80.

[20] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An ASIC flow for GHz asynchronous designs," *IEEE Design and test of Computers*, vol. 28, no. 5, pp. 36–51, 2011.

[21] A. Saifhashemi, D. Hand, P. Beerel, W. Koven, and H. Wang, "Performance and area optimization of a bundled-data intel processor through resynthesis," in *Asynchronous Circuits and Systems (ASYNC), 2014 20th IEEE International Symposium on*, May 2014, pp. 110–111.

[22] D. Hand, M. T. Moreira, H.-H. Huang, D. Chen, F. Butzke, Z. Li, M. Gibiluka, B. M., C. N. L. V., and P. A. Beerel, "Blade - a timing violation resilient asynchronous template," in *ASYNC*, May 2015.

[23] https://github.com/AmeerAbdelhadi, 2018.