

# Flexible Active–Passive and Push–Pull Protocols

Ebelechukwu Esimai<sup>ID</sup>, *Graduate Student Member, IEEE*, and Marly Roncken<sup>ID</sup>, *Member, IEEE*

**Abstract**—By means of a simple buffer design, we show that *active versus passive* and *push versus pull* settings in asynchronous communication protocols, also known as handshake protocols, can be controlled by initialization. We advocate postponing initialization until run time and show that postponement simplifies the design and design process and serves test, debug, and analysis. We design the buffer as a network of communication channels with storage, called *Links*, and storage-free computation modules, called *Joints*. We describe the behaviors of Links and Joints using a *shared variable model* presented here for the first time.

**Index Terms**—Asynchronous circuit, communication protocol, distributed system, initialization, late binding, Link–Joint model.

## I. INTRODUCTION

THIS letter focuses on the use of *active* versus *passive* and *push* versus *pull* in asynchronous communication protocols, also known as handshake protocols. We are particularly interested in communication protocols generated as part of a large distributed system with automated support of a syntax directed or otherwise structured compiler [1]–[3], [6], [10].

We specify such systems as networks of communication channels with storage, called *Links*, and computation modules without storage, called *Joints* [7]–[9]. We prefer Link–Joint systems because they enable us to postpone implementation decisions. Links and Joints cover most if not all asynchronous protocols and circuit families.

Active and passive apply to a Link’s ends, also called *ports*. Push and pull apply to the entire Link. Specifically:

- 1) an *active* Link port starts the communication;
- 2) a *passive* Link port responds;
- 3) a Link *pushes* data from its active port to passive port;
- 4) a Link *pulls* data from its passive port to active port;
- 5) a bidirectional Link pushes and pulls data.

Active–passive and push–pull protocol settings determine whether the system is functional and how well it performs. How to optimize protocol settings for best performance is outside the scope of this letter. Instead, this letter shows that we can implement any active–passive and push–pull protocol settings that a designer or a compiler may assign to a system of Links and Joints *merely by initializing Link storage*!

Manuscript received 11 November 2021; revised 22 January 2022 and 7 March 2022; accepted 10 March 2022. Date of publication 15 March 2022; date of current version 23 August 2022. This work was supported in part by private sponsors through the Portland State University Foundation. This manuscript was recommended for publication by S. Katkooi. (Corresponding author: Ebelechukwu Esimai.)

The authors are with the Asynchronous Research Center and the Computer Science Department, Maseeh College of Engineering and Computer Science, Portland State University, Portland, OR 97201 USA (e-mail: esimai@pdx.edu; mroncken@pdx.edu).

Digital Object Identifier 10.1109/LES.2022.3159492

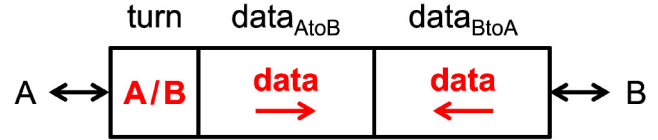


Fig. 1. A Link shares three variables, *turn*,  $data_{AtoB}$ , and  $data_{BtoA}$ , between its two distinct ports, *A* and *B*. Variable *turn* identifies “whose turn it is” to update the Link variables—*A*’s or *B*’s. Variables  $data_{AtoB}$  and  $data_{BtoA}$  contain the data going from *A* to *B* and from *B* to *A*, respectively. Unless one or both bit vectors for  $data_{AtoB}$  and  $data_{BtoA}$  have zero bit width, the Link is bidirectional as indicated by the bidirectionality of the arrows at ports *A* and *B*.

In other words, the same design *can* serve multiple settings—yet, many designs *do not* and suffer unnecessary complexity, because their designers bind settings prematurely.

- 1) Compilers and logic designers often fix active–passive and push–pull settings [1]–[3], [6], creating multiple design and validation tasks where one would suffice.
- 2) Circuit designers often build a fixed setting into the circuit either directly or with a reset signal [10, Fig. 9.8], which may complicate both the design and its testability.

We use the example of a first-in–first-out buffer (FIFO) to illustrate these points. Section II introduces Links and Joints and builds up the example. Section III uses the FIFO example to set active–passive and push–pull by initialization. Section IV advocates for freedom of initialization right up to run time. Section V concludes this letter.

## II. PREPARING THE EXAMPLE

We use a network of alternating Links and Joints to build the example for this letter—a FIFO. Sections II-A and II-B describe the functional behaviors of Links and Joints using a shared variable model. This new model is simpler but the behaviors it specifies are consistent with our earlier specifications [7]–[9] and model [4]. Section II-C presents the Link and Joint design of the FIFO.

### A. Shared Variable Model for Links

A Link stores and transports data and control information between its two distinct ends, *A* and *B*, also called *ports*. In this letter, we ignore the transport part of a Link and model the storage part as three shared variables, *turn*,  $data_{AtoB}$ , and  $data_{BtoA}$ —see Fig. 1. Variable *turn* designates the port to update the Link variables. When *turn* designates *A*, denoted as “*turn* = *A* is valid,” port *A*—ultimately representing a Joint or environment connected to *A*—may update *turn* and  $data_{AtoB}$ . Specifically, port *A* may change *turn* to designate *B* and may update  $data_{AtoB}$  with new data going from *A* to *B*. Likewise, when *turn* = *B*, port *B* may update *turn* and  $data_{BtoA}$ . Data

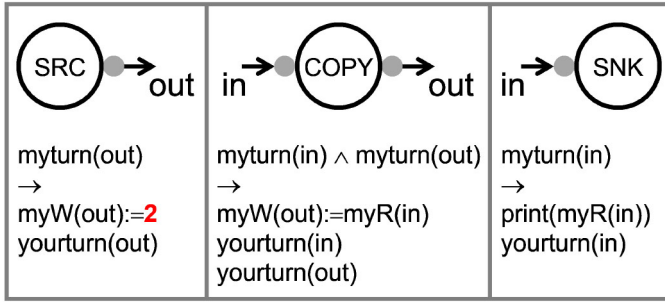


Fig. 2. Joints SRC, COPY, and SNK with formal Link port names (top) and guarded command specifications (bottom). Data flow from left to right in the direction of the arrows. Each port is marked with a small circle, colored gray here to indicate that Link variable *turn* has yet to be initialized.

may be typed, but for the purpose of this letter it suffices to model  $\text{data}_{AtoB}$  and  $\text{data}_{BtoA}$  as bit vectors. When none, exactly one, or both data bit vectors have zero bit width, we consider the Link bidirectional, unidirectional, or data-less, respectively. Henceforth, we will refer to the individual ports and variables of Link  $l$  as  $l.A$ ,  $l.B$ ,  $l.\text{turn}$ ,  $l.\text{data}_{AtoB}$ , and  $l.\text{data}_{BtoA}$ .

### B. Extending the Shared Variable Model to Joints

A Joint computes and controls the information flow between Link ports to which it connects. One can view a Joint as a place where Links meet to exchange information. In particular, a Joint may exchange information between both ports of the same Link [8]. Joint specifications use the following terms to access Link variables from port  $p$  of Link  $l$ , where parameter  $p$  is either  $l.A$  or  $l.B$ .

- 1)  $\text{myturn}(p)$ : A Boolean that indicates if it is  $p$ 's turn to update the shared variables in Link  $l$ . Denotes " $l.\text{turn} = A$ " if  $p = l.A$ , and " $l.\text{turn} = B$ " if  $p = l.B$ .
- 2)  $\text{yourturn}(p)$ : An assignment that relinquishes the turn to update  $l$ 's variables from  $p$  to  $l$ 's other port. Denotes " $l.\text{turn} := B$ " if  $p = l.A$ , " $l.\text{turn} := A$ " if  $p = l.B$ .
- 3)  $\text{myR}(p)$ : A data bit vector stored by Link  $l$  and intended to be read by  $p$ —that is, by the Joint connected to  $p$ . Denotes  $l.\text{data}_{BtoA}$  if  $p = l.A$ ,  $l.\text{data}_{AtoB}$  if  $p = l.B$ .
- 4)  $\text{myW}(p)$ : A data bit vector stored by  $l$  and intended to be written by  $p$ —that is, by the Joint connected to  $p$ . Denotes " $l.\text{data}_{AtoB}$ " if  $p = l.A$ , " $l.\text{data}_{BtoA}$ " if  $p = l.B$ .

Given that Link port  $p$  only reads and never writes  $\text{myR}(p)$ ,  $\text{myR}(p)$  is stable as long as  $\text{myturn}(p)$  holds because then Link  $l$ 's other port lacks the turn to change the Link variables. The terms,  $\text{myturn}(p)$ ,  $\text{yourturn}(p)$ ,  $\text{myR}(p)$ , and  $\text{myW}(p)$ , exploit the symmetry of Links and allow a Joint's formal specification to be silent as to which end of a Link  $p$  is at.

Fig. 2 shows three Joints: SRC, COPY, and SNK. We draw a Joint as a big circle containing the Joint name and attach a small circle for each Link port to which the Joint connects. Each Link port has a formal name defined only in the scope of the Joint and used in the formal specification of the Joint. With each port, we draw an arrow to indicate which way Link data flow. All Link ports in Fig. 2 are unidirectional, with Link data entering a Joint at a port with formal name *in* and leaving a Joint at a port with formal name *out*.

We use guarded commands [5] to specify the behavior of each Joint [8] and an interleaving model to combine behaviors. Each Joint in Fig. 2 has a single guarded command, with the Boolean guard separated by an arrow ( $\rightarrow$ ) from the command. Each Joint must wait until its guard is *true* before it may execute its command. Command execution is atomic: changes to Link variables become visible atomically, i.e., all at once, upon termination of the command's execution.

In Fig. 2, each Joint waits to execute its command until all its Link ports have their turn. During execution:

- 1) Joint SRC writes the bit vector for value 2 to  $\text{myW(out)}$  and relinquishes *out*'s turn;
- 2) Joint COPY copies data from  $\text{myR(in)}$  to  $\text{myW(out)}$  and relinquishes both *in*'s turn and *out*'s turn;
- 3) Joint SNK prints the data value stored in  $\text{myR(in)}$  and relinquishes *in*'s turn.


Fig. 2 refrains from specifying whether the commands execute their statements in sequence or in parallel. Both work, here, because: 1) each command statement changes a different Link variable and 2) command execution is atomic.

### C. Example: FIFO

FIFOs are essential for bridging differences in throughput between a source (SRC) and sink (SNK) and for optimizing overall system throughput. In Fig. 3(a)–(d), we build four FIFO configurations by “linking” instances of Joints SRC, COPY, and SNK in series. With the instance of Joint COPY, the two Links,  $L_1$  and  $L_2$ , form a two-stage FIFO that can store zero, one, or two data items between SRC and SNK. To optimize throughput, FIFOs may be primed with initial data and control information.

### III. PRIMING THE FIFO WITH DATA AND CONTROL

The four FIFO configurations in Fig. 3(a)–(d) offer different initializations for data and control variables in Links  $L_1$  and  $L_2$ . Each Link stores a single control variable, called *turn* in Fig. 1, and—being unidirectional—a single data variable of nonzero bit width, called  $\text{data}_{AtoB}$  or  $\text{data}_{BtoA}$  depending on whether the data stored in it go from Link port A to B or vice versa. To distinguish changes in *turn* from changes in data when “running” the FIFO, Fig. 3 uses:

- 1) a pin at the Link port designated by *turn* (  ), attaching permission to change the Link variables to that port;
- 2) arbitrary (\*) or specific numeric values for data;
- 3) a mere arrow to depict a Link.

Data in Fig. 3 flow from left to right in the direction of the arrows. A Link is considered empty and without relevant data, if the Link port that writes the data has the turn. A Link is considered full and has relevant data if the Link port that reads the data has the turn. We call the FIFO in Fig. 3 empty if both Links  $L_1$  and  $L_2$  are empty, full if both Links are full, and half full if one Link is empty and the other Link is full. The four FIFO configurations in Fig. 3 are initially empty (a-top), full (b-top), and half full (c-top, d-top).

Fig. 3(a)–(d) run the same FIFO design! The red-colored pins and data values in Fig. 3(a-top)–(d-top) are part of the initial state, as are the active (●) and passive (○) port settings explained in more detail in Section III-A.

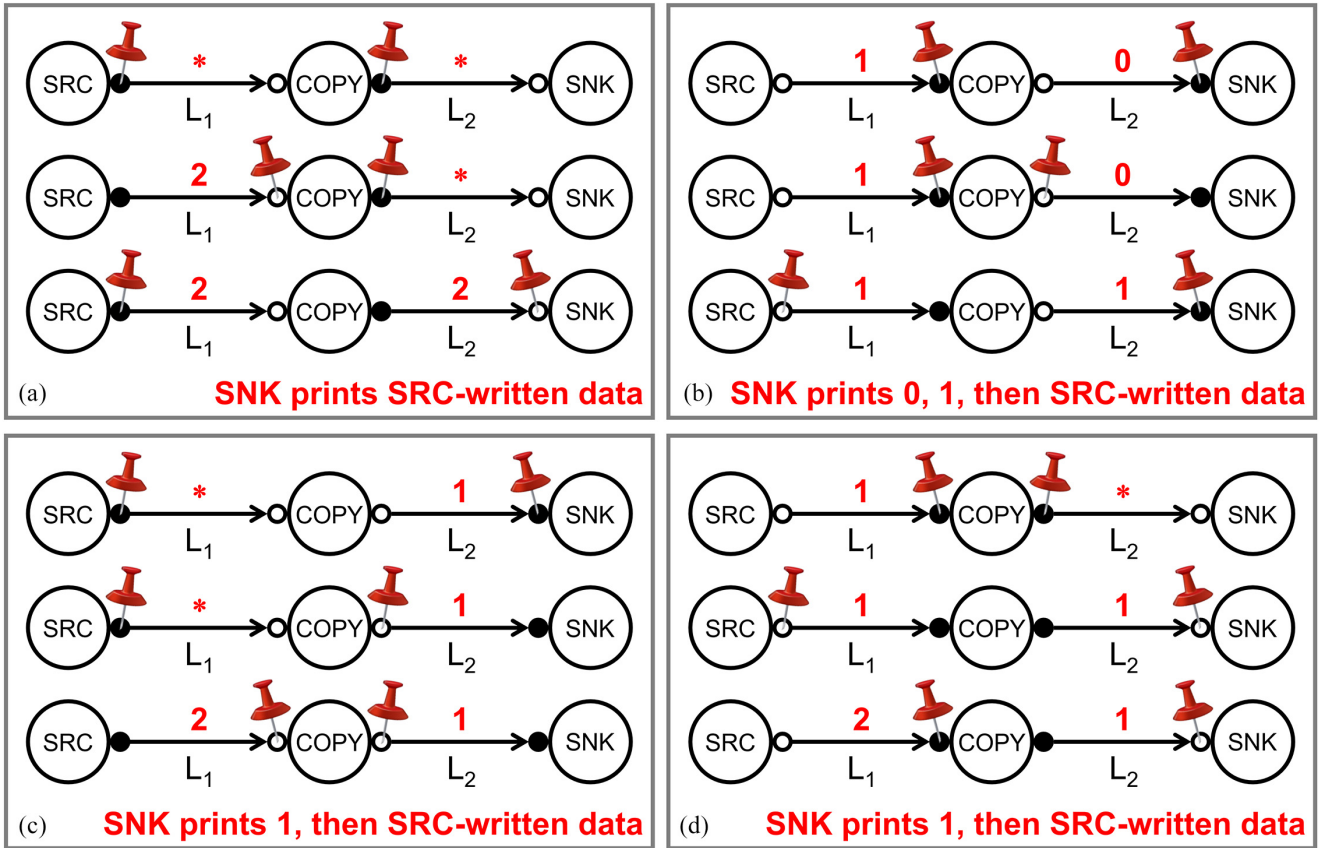


Fig. 3. Panels (a)–(d) each show three states in an interleaving model running a two-stage FIFO between SRC and SNK. Data flow from left to right, as specified in Figs. 1 and 2. Each run starts at the top row and proceeds to the next lower row in the panel. The runs start from a FIFO that is initially (a) empty, (b) full, or (c) and (d) half full. Instead of the gray ports in Fig. 2, the panels use opaque black circles (●) for active ports and transparent black circles (○) for passive ports. Note that each panel uses a different active versus passive port setting.

- In an initially empty FIFO with irrelevant arbitrary Link data (top), Joint COPY waits until Joint SRC has written a new data value, 2, into Link  $L_1$  and relinquished its turn on  $L_1$  (middle). Joint COPY (middle) now has all the Link turns necessary to act. It acts by copying  $L_1$ 's data value, 2, to  $L_2$  and by relinquishing both Link turns (bottom). The FIFO is now half-full with relevant data in  $L_2$ .
- In an initially full FIFO (top), Joint COPY waits until Joint SNK has read and printed  $L_2$ 's data value, 0, and relinquished its turn on  $L_2$  (middle). Joint COPY (middle) now has all the Link turns necessary for it to act. It acts by copying  $L_1$ 's data value, 1, to  $L_2$  and by relinquishing its turn on Link  $L_1$  and its turn on Link  $L_2$  (bottom). The FIFO is now half-full with relevant data in  $L_2$ .
- In an initially half-full FIFO with the relevant data in  $L_2$  (top), Joint COPY waits until SNK has printed  $L_2$ 's data value, 1, and relinquished its turn on  $L_2$  (middle). COPY also waits until SRC has written a new data value, 2, into Link  $L_1$  and relinquished its turn on  $L_1$  (bottom). Though SNK acts before SRC here, they may act in either order. The FIFO is now half-full with relevant data in  $L_1$ .
- In an initially half-full FIFO with relevant data in  $L_1$  (top), Joint COPY has all the Link turns necessary to act. It acts by copying  $L_1$ 's data, 1, to  $L_2$  and by relinquishing both Link turns (middle). The FIFO (middle) is now half-full with the relevant data in  $L_2$  and with COPY waiting for SNK and SRC to act, as in Fig. 3(c-top). This time, SRC acts first (bottom).

We initialized FIFO data such that Joint SNK may print consecutive numbers up to and including the first SRC-written data value 2. Fig. 3(a)–(d) each show three consecutive states of a running FIFO, including the initial state. As expected, SNK prints initial FIFO data—none for (a), 0 then 1 for (b), 1 for (c) and (d)—and then SRC-written data.

Running FIFOs follow the Link-Joint protocol, where Joints take turns reading and writing Link data by “playing ping-pong” with a Link’s *turn*. This protocol has the following practical consequences.

- 1) The FIFO never uses data stored in initially empty Links.
- 2) Only data in initially full Links require initialization.
- 3) Links can retain an old data or control value until a new one replaces it—clearance is unnecessary.

#### A. Active-Passive and Push-Pull Are Set by Initialization

All *turn* variables in Fig. 3 are initialized. So, rather than the gray-colored circles in Fig. 2 that mark ports of Links with an uninitialized *turn* variable, Fig. 3 uses different markings.

- 1) Fig. 3 uses opaque black circles (●) for ports that initially have the turn. We call these ports *active*.
- 2) Fig. 3 uses transparent black circles (○) for ports initially lacking the turn. We call these ports *passive*.
- 3) In Fig. 3, Links with an active port on the left and a passive port on the right *push* data from left to right.
- 4) In Fig. 3, Links with a passive port on the left and an active port on the right *pull* data from left to right.

Our use of opaque and transparent black circles in Fig. 3 and the terms active versus passive and push versus pull agree

with notations and terminology introduced and used in prior research at Caltech [3], Philips Research [2], [6], and the University of Manchester [1].

In the top row of Fig. 3(a), the  $L_1$  port connected to SRC and the  $L_2$  port connected to COPY are active. Note that the two ports remain active while the FIFO state progresses to lower rows in Fig. 3(a). Note that the two other ports remain passive. Consequently, given that the data in Fig. 3(a) keep flowing from left to right in the direction of the arrows,  $L_1$  and  $L_2$  push data initially and keep pushing data while the FIFO runs—until we reinitialize the system.

We can reinitialize the FIFO as indicated in Fig. 3(b-top). The  $L_1$  port connected to COPY and the  $L_2$  port connected to SNK then become active, the ports previously active in (a) then become passive, and  $L_1$  and  $L_2$  then pull data.

#### IV. FREEDOM OF INITIALIZATION

When priming a FIFO, a designer or a compiler tends to fill the FIFO up to a specific level with specific data values and—depending on latency and throughput requirements—with a specific distribution of full versus empty Links. Fig. 3, therefore, focuses primarily on filling levels and data contents of the FIFO and provides two initial configurations for a half-full FIFO to hint at potential differences in latency.<sup>1</sup>

Fig. 3 illustrates that the same design serves multiple active-passive and push-pull settings if we postpone initialization and bind settings late. Rather than binding settings at compile or fabrication time, we can wait until run time. We can even bind some system parts early and others late. Compared to systems that use “fixed protocol settings” [1]–[3], [6], flexibility to bind protocol settings late offers the following benefits.

- 1) *Small Library*: Our Link and Joint model specifies basic system parts independent of an initial state these may assume, resulting in *one* Link specification in Fig. 1 and *three* Joint specifications in Fig. 2. Had we used “fixed protocol settings,” Fig. 1 would require *two* channels and Fig. 2 *eight* modules, each with a different specification.
- 2) *Simple Design Process*: Our Link and Joint model combines basic parts into larger designs and ultimately into systems, independent of initial states these may assume. The initial values of the Link variables in Fig. 3(a) differ from those in Fig. 3(b)–(d) but the four FIFO designs are identical. Had we used “fixed protocol settings,” Fig. 3 would require *not one* but *four* different FIFO designs.
- 3) *Test Compatible*: Effective test, debug, and analysis often require protocol settings other than those used for normal operation [9]. Late binding makes initialization for test runs as easy as initialization for normal runs. A circuit with “fixed protocol settings” in need of other settings for test would require bypass features, complicating circuit or test.<sup>2</sup>

<sup>1</sup>Latency and throughput evaluations require a real-time model rather than the interleaving model that we use in this letter.

<sup>2</sup>Fig. 9.8 in the book by Jens Sparsø [10] shows a Micropipeline circuit implementation with fixed protocol settings for a 2-stage half-full ring FIFO.

Neither early nor late bindings remove the need to know which initial states a system should use to run as intended. This knowledge is necessary to analyze and guarantee crucial correctness and performance metrics of the system. Late binding has the advantage that it can support test metrics too, paving the way for high-level test generation.

We created Links and Joints to have a unified design and test approach for asynchronous systems, regardless of circuit family and technology [9]. Using this approach, we can “program” protocol settings by connecting Link variables to a general access mechanism. We also connect Joint guards by adding accessible *go* signals to safely stop, (re-)initialize, and start executions. The Joint specifications in Fig. 2 implicitly assume the presence of *go* signals. “Programmable” protocols, currently unique to Link-Joint designs [7]–[9], can be adopted by any high-level asynchronous design approach.

#### V. CONCLUSION

Asynchronous designs often suffer unnecessary complexity because logic designers, circuit designers, or compilers bind protocol settings prematurely. Flexible binding of settings as well as of choice of protocol and even circuit family gives Link and Joint systems enjoyable simplicity in design, design process, and even test, debug, and analysis. We encourage others to adopt a similarly flexible design approach.

#### ACKNOWLEDGMENT

The authors thank Gary Delp, Bart McCoy, Warren Hunt, Jr., and Ivan Sutherland for support and productive research discussions.

#### REFERENCES

- [1] A. Bardsley, “Implementing Balsa handshake circuits,” Ph.D. thesis, Dept. Comput. Sci., Univ. Manchester, Manchester, U.K., 2000.
- [2] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schali, and A. Peeters, “Asynchronous circuits for low power: A DCC error corrector,” *IEEE Design Test Comput.*, vol. 11, no. 2, pp. 22–32, 1994.
- [3] S. M. Burns, “Automated compilation of concurrent programs into self-timed circuits,” M.S. thesis, Dept. Comput. Sci., California Inst. Technol., Pasadena, CA, USA, 1987.
- [4] C. Chau, W. A. Hunt, Jr., M. Kaufmann, M. Roncken, and I. Sutherland, “A hierarchical approach to self-timed circuit verification,” in *Proc. IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, 2019, pp. 105–113.
- [5] E. W. Dijkstra, “Guarded commands, nondeterminacy and formal derivation of programs,” *Commun. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.
- [6] A. Peeters, F. te Beest, M. de Wit, and W. Mallon, “Click elements: An implementation style for data-driven compilation,” in *Proc. IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, 2010, pp. 3–14.
- [7] M. Roncken and I. Sutherland, “Design and test of high-speed asynchronous circuits,” in *Asynchronous Circuit Applications*, J. Di and S. C. Smith, Eds. London, U.K.: Inst. Eng. Technol. (IET), 2020, ch. 7, pp. 113–171.
- [8] M. Roncken *et al.*, “How to think about self-timed systems,” in *Proc. Asilomar Conf. Signals Syst. Comput.*, 2017, pp. 1597–1604.
- [9] M. Roncken, S. Mettala Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, “Naturalized communication and testing,” in *Proc. IEEE Int. Symp. Asynchronous Circuits Syst. (ASYNC)*, 2015, pp. 77–84.
- [10] J. Sparsø, *Introduction to Asynchronous Circuit Design*. Kongens Lyngby, Denmark: DTU Compute, 2020.