2015 21st IEEE International Symposium on Asynchronous Circuits and Systems

Naturalized Communication and Testing

Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland
Asynchronous Research Center
Maseeh College of Engineering & Computer Science, Portland State University, Portland, Oregon, USA
marly.roncken@gmail.com, mettalag@cecs.pdx.edu, parkhoon@gmail.com, navp@pdx.edu, clcowan@cecs.pdx.edu, ivans@cecs.pdx.edu

***Abstract*—We "naturalize" the handshake communication links of a self-timed system by assigning the capabilities of filling and draining a link and of storing its full or empty status to the link itself. This contrasts with assigning these capabilities to the joints, the modules connected by the links, as was previously done. Under naturalized communication, the differences between Micropipeline, GasP, Mousetrap, and Click circuits are seen only in the links — the joints become identical; past, present, and future link and joint designs become interchangeable.**

We also "naturalize" the actions of a self-timed system, giving actions status equal to states — for the purpose of silicon test and debug. We partner traditional scan test techniques dedicated to *state* with new test capabilities dedicated to *action*. To each and every joint, we add a novel proper-start-stop circuit, called MrGO, that permits or forbids the action of that joint. MrGO, pronounced "Mister GO," makes it possible to (1) exit an initial state cleanly to start circuit operation in a delay-insensitive manner, (2) stop a running circuit in a clean and delay-insensitive manner, (3) single- or multi-step circuit operations for test and debug, and (4) test sub-systems at speed.

I. Introduction

Point of view is worth 80 IQ points
Alan Kay, Turing Award 2003, Kyoto Prize 2004, Draper Prize 2004

We view a self-timed dataflow or pipeline system as a directed graph with links as edges and joints as nodes, as suggested by Figure 1. The links are the communication channels, with data flowing in the direction of the arrows. The joints are the self-timed modules that implement flow control and data operations. In this paper, we take a novel point of view of links and joints. We present this view using two-phase handshake channels with bundled data [9] as links, and Micropipeline [11], GasP [12], Mousetrap [8], and Click [5] modules as joints. Note however that this new viewpoint is useful beyond these self-timed families and protocols!

Links deserve the full attention of circuit designers because they consume most of the energy, cause most of the delay, and occupy most of the area in a modern digital system. Nevertheless, publications presenting the Micropipeline, GasP, Mousetrap, and Click circuit families treat the links merely as simple wires and put all the digital logic in the joints. We offer a different, communication-aware or link-aware, point of view. Our link-aware view puts equal emphasis on links and joints, by giving each the digital logic needed to perform its role in the system. The title of the paper comes from the idea that naturalized [1] citizens share the same rights as native-born citizens. We naturalize links, giving them status equal to joints.

[1]Although there are many two-word oxymorons, such as the "guest host" for a late night TV show or a "giant shrimp," single-word oxymorons, like "naturalized," are rare.

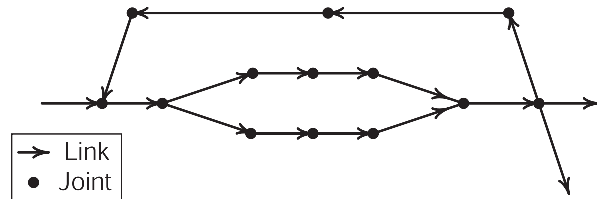

Figure 1 A self-timed dataflow system with communication channels, called links, and flow control and data computation modules, called joints, can be viewed as a directed graph with data flowing in the direction indicated by the arrows.

This point of view is so simple that readers may consider it obvious. Simple, yes, but it is also very powerful for it reveals how self-timed circuits and systems work, how to represent them, how to design them, and how to test them. This point of view unifies the existing families of self-timed circuits.

The role of a joint becomes much clearer and much simpler after pruning away its link-specific tasks. Joints are, more obviously than ever, the meeting points for links to coordinate states and exchange data. The coordinating actions are done in the joints, making joints the ideal place to start and stop self-timed action. This sets the stage for a new view of testing, with joints controlling the actions and links holding the states.

To support this test view, we advocate adding a novel proper start-stop circuit, called MrGO, to each and every joint in the system. MrGO, pronounced "Mister GO," has a single external input, called *go*, which it arbitrates against pending or underway joint actions. De-assertion of the *go* signal to MrGO provides reliable stopping of self-timed operation, but more importantly, freezes joint action. Freezing joints while initializing links to full or empty prevents the self-timed joint actions from prematurely changing the initial states of links. Selectively permitting joints to "go" allows for single- and multi-step operation and at-speed testing of sub-systems. MrGO removes the timing uncertainty of every joint, thus rendering a desired part or all of a self-timed system as orderly as a clocked system, whenever needed.

The outline of this paper is as follows. Section II reviews the Micropipeline, GasP, Mousetrap, and Click circuit families, as presented in their original publications. Section III presents the new link-aware view and the corresponding link-joint interface for "what a link tells a joint" and "what a joint tells a link." Section IV presents test and debug from the new point of view and an implementation for MrGO and its use. Section V describes test scenarios with MrGO performed on two 40nm TSMC chip experiments. Section VI concludes the paper.

1522-8681/15 $31.00 © 2015 IEEE
DOI 10.1109/ASYNC.2015.20

IEEE computer society

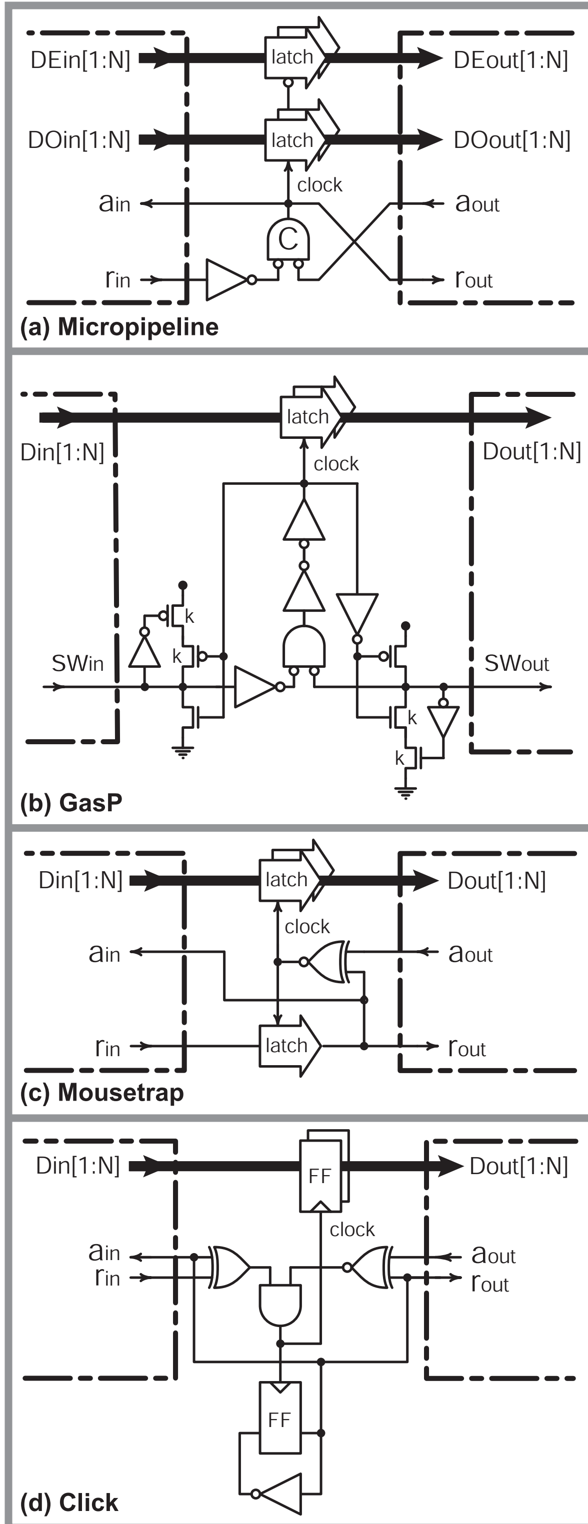

Figure 2 Circuit designs for a single-input single-output FIFO module in each of the four bundled-data two-phase circuit families. The dashed lines mark the interface between the joint in the middle and the left-hand and right-hand links. Note that the joints hold all the logic for flow control and data computation — the links are "just wires."

II. Four Bundled-Data Two-phase Circuit Families

The bundled-data circuit descriptions for Micropipeline [11], GasP [12], [10], Mousetrap [8], and Click [5] use two-phase handshake protocols to communicate the presence or absence of valid data on a link. A *full* link carries valid data. An *empty* link carries data that are no longer or not yet significant. Figure 2 describes the circuit of a FIFO module in each family.

The circuit description of each FIFO module includes half of the incoming link over which data arrive, a joint between the two links, and half of the outgoing link. The designs emphasize the joint between links, and treat each link as merely a handshake channel of nothing but wire. This joint-centric view has implications for initialization and testing at the system level — see Section III-C. To restore compositionality for initialization and testing, we re-designed the four circuit descriptions from a link-aware point of view.

Before we explain the re-designs, let us go over the original module designs in Figure 2. Whenever the incoming link is full and the outgoing link is empty, the FIFO module performs three tasks:

- capture and hand-over one data item,
- make the incoming link empty, and
- make the outgoing link full.

These tasks are performed in parallel. They are repeated when the incoming link has new data and is again full and the outgoing link has transferred captured data and is again empty. The four FIFO module designs differ mainly in two ways:

- how they represent full and empty links, and
- when they capture data.

The representations for full and empty links depend on the specific variant of the two-phase handshake protocol used. Micropipeline, Mousetrap and Click use a non-return-to-zero (non-RTZ) variant. GasP uses a return-to-zero (RTZ) variant. The link representations used in this paper appear in Figure 3.

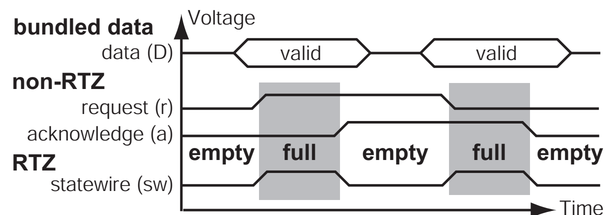

Figure 3 Non-RTZ and RTZ two-phase handshake variants. By convention, data must be valid when the link is full, and may change only when the link is empty. In reality, data may be kited, as long as the values are valid when captured.

When data are captured depends on the protocol variant for bundled data. Micropipeline and Mousetrap use *normally-transparent* latches for which the clock signal is high when the outgoing links that forward the data are empty. The intent is to decrease latency by forwarding data as far into the pipeline as is possible. GasP and Click use *normally-opaque* latches and flipflops for which the clock signal is high when all incoming links over which the data arrive are full and all outgoing links that forward the data are empty. The intent is to save energy by preventing data from rippling through the pipeline prematurely.

These family differences, explained for a simple FIFO design but present in any design at the module- or system- level, make the various circuit families much harder to work with than is necessary, and much harder to exchange or combine. These differences permeate many parts of a design flow, ranging from compilation and throughput analysis to relative timing verification, static and dynamic timing and function validation, and silicon test and debug.

In Section III, we present a link-aware re-design approach that de-emphasizes the differences in "how data are captured" and "how full and empty links are represented" by moving both the data latches and the full and empty logic out of the joints and into the links. This results in a standard link-joint interface for all four circuit families, and allows free exchange between the families of past, present, and future link and joint designs.

III. NATURALIZED COMMUNICATION

Figure 4 repeats the GasP FIFO module of Figure 2(b) after moving both the data latches and the full and empty link retention into the links. The interface signals thus created sense the full-empty status of the links ($full_{in}$, $full_{out}$), hand over data (D_{in}, D_{out}), make an incoming link empty ($drain_{in}$), and an outgoing link full ($fill_{out}$).

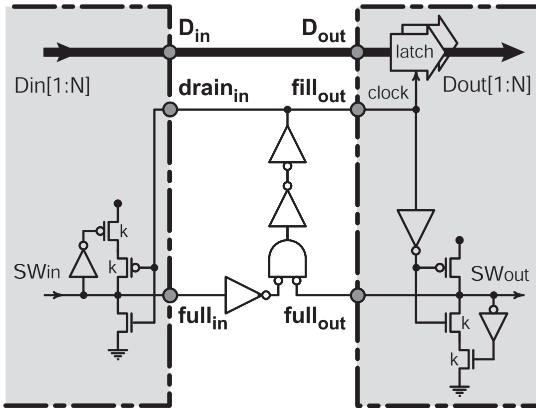

Figure 4 Circuit re-design for the GasP FIFO module of Figure 2(b). The responsibility for capturing data has moved from the joint to the receiving link. Representing full or empty is now entirely in the links. We colored "link logic" grey, reserving the white center for the simplified joint.

One forms longer GasP FIFOs by connecting GasP FIFO modules head to tail. Each head-to-tail connection pairs the "link logic" in the two grey halves, forming a closed grey rectangle as in Figure 5. Inside each GasP grey rectangle live data latches, a statewire, weak half-keepers, and strong pull-up and pull-down drivers. Outside, we see D, $fill$, $drain$, and $full$. The GasP control circuitry for the link in Figure 5, which we call a *naturalized link*, appears in Figure 6(b).

Our new viewpoint makes the link-joint interface of Figure 5 the standard communication interface for all four families. The interface involves data and commands from the joint and state reports from the link. The joint can command a full incoming link to drain and an empty outgoing link to fill. Fill and drain

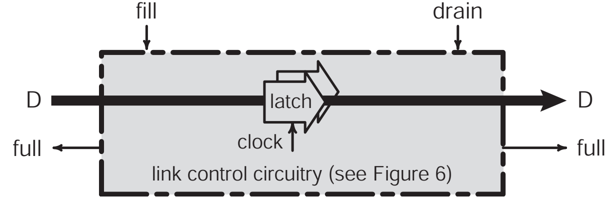

Figure 5 Naturalized link with standard link-joint interface. Wiring clock to $fill$ makes the latches normally-opaque, and wiring clock to $\neg full$ makes them normally-transparent.

commands come from opposite ends of the link. Data flow from one end of the link to the other end, and are captured in-between. Each link reports its full-empty state to the joints at its two ends. Those reports, as well as the data at the two ends, may differ briefly as information flows through the link.

In the following sub-sections, we discuss the types of links (Section III-A) and the types of joints (Section III-B) that the naturalized communication viewpoint yields, and their impact on self-timed system design (Section III-C).

A. Link Types

A naturalized link receives fill or drain commands and data. It reports the data and its full-empty state. Data flow from one end of the link to the other end, and are captured in-between. Fill and drain commands arrive at opposite ends of the link. When the link receives a fill command, i.e. *fill* is high, the link changes its state to full. Upon receiving a drain command, i.e. *drain* is high, the link changes its state to empty.

Data may flow normally-opaque or normally-transparent in each link, as indicated in Figure 5. Fill and drain actions can be implemented in various ways. Figure 6 shows a few options:

- The GasP link in Figure 6(b) has two isolated transistors, one at each end of the link, to perform the fill and drain actions: the PMOS pull-up transistor fills; the NMOS pull-down transistor drains. The statewire itself and the two half-keepers, one at each end, maintain the full-empty state between fill and drain actions. The states at the two ends may differ briefly but will ultimately match [2].
- Each Micropipeline (a), Mousetrap (c), and Click (d) link stores the full-empty state on two wires: the request wire and the acknowledge wire. The fill action changes the request wire, making its value differ from that of the acknowledge. The drain action changes the acknowledge wire, making its value match that of the request. Exclusive-OR gates generate the full-empty state of the link by comparing the request and the acknowledge values, giving full (1) for "differ" and empty (0) for "match." Because request and acknowledge are separate signals, they have separate state-holding circuit elements to hold and change them. Micropipeline and Mousetrap change each wire by copying the other wire and complementing if needed. Click changes each wire by complementing its value, using a flipflop to store the old and new values.
- A Set-Reset flipflop (e) provides a simple and perhaps the most standard link control circuit one can imagine.

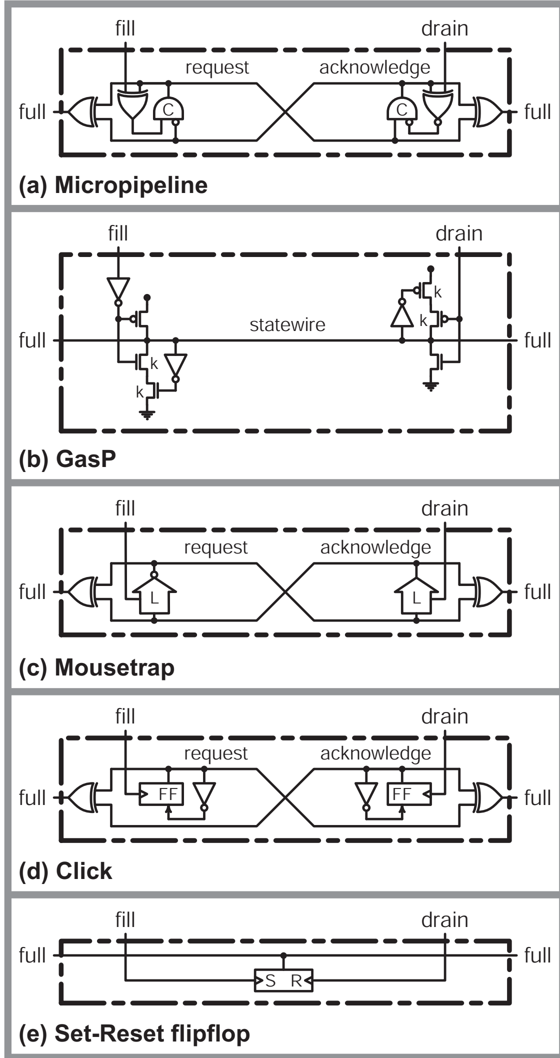

Figure 6 Implementation examples of link control circuitry. These supplement the link data circuitry of Figure 5.

B. Joint Types

Joints respond to the full and empty state of their links. In general, the control logic of a joint is an AND-function of the conditions necessary for it to act. Some joints have multiple such AND-functions to guard different actions. The response of a joint usually changes one or more of the link states to which the joint responded. Thus, there is a feedback loop from link-state to joint-action and back to link-state. The throughput of a self-timed system is in part dependent on the delay of such feedback loops. The delay may be adjusted to accommodate data operations coordinated by the joint. The signals that call for fill or drain action may persist for only a short time, a time whose duration depends on the circuits in the feedback loop.

Naturalizing the links clarifies the role of a joint. Joints that pipeline, fork, or join combinational dataflow operations can be free of stored state. Figure 7 sketches the design of such a joint. The joint in Figure 4 (center section) is an example of such a joint — with $n = m = 1$.

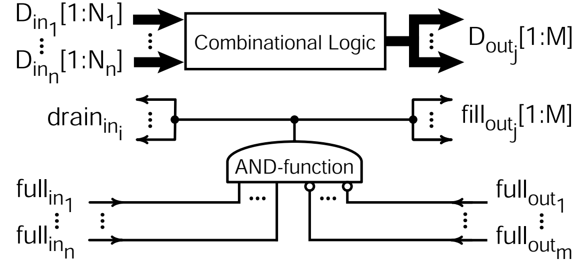

Figure 7 Design sketch of a joint without stored state with n naturalized incoming and m naturalized outgoing links, where $1 \leq i \leq n$ and $1 \leq j \leq m$. If data are just copied then $D_{out_j}[1:M]$ is the concatenation of $D_{in_1}[1:N_1]$ to $D_{in_n}[1:N_n]$ and $M = N_1 + \ldots + N_n$ $(N_1, \ldots, N_n \geq 0)$.

Stored state for control logic appears only in joints sensitive to selective link participation. Examples are joints that send arriving data alternately to different outgoing links, joints that arbitrate between incoming links, and joints that guard the participation of links based on data reported on other links.

The store-free joint shown in Figure 7 works with any of the links in Figure 6. The same is true for joints with locally stored state for flow control. The functionality and compositionality of each link combination are the same. Different links may have different timing constraints — a topic that, due to space limitations, is outside the scope of this paper. However, the fact that functional and timing differences are confined to the links simplifies modeling, validation, and silicon compilation.

With the responsibilities for full-empty link retention and data storage assigned to the links, the link-aware view makes both types of joints significantly easier to understand and design.

C. Impact of Naturalization on System Design

The link-aware point of view offers complete generality to self-timed systems. All types of links are interchangeable — see Figure 6. Moreover, substituting a normally-transparent for a normally-opaque link or vice versa is always possible — see Figure 5. System designers can choose which type to use based on system demands for power conservation or data latency. Remarkably, the circuit of Figure 7 can drive a mix of normally-transparent and normally-opaque outgoing links.

Figure 8 illustrates the impact on throughput that naturalized communication can have. The naturalized Mousetrap ring with Mousetrap links is slower than the original Mousetrap ring. However, the naturalized Mousetrap ring with GasP links is faster than the original Mousetrap ring.

Throughput differences between naturalized and original pipelines within the same family become smaller and may disappear completely for joints that accommodate selective participation. This is because selective participation and shared state do not fare well together, and because AND-functions and exclusive-OR gates that can be optimized away when *all* links participate become essential when it is necessary to *select* participants — as the original Click modules in [5] attest.

We avoided estimating the power and area cost of naturalized communication, because we expect that power and area are dominated by datapath operations and wire lengths.

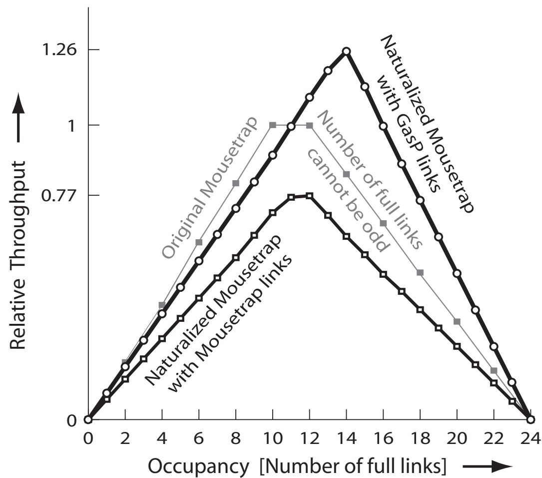

Figure 8 Three canopy graphs for simulations of rings with 24 pipeline stages. The center graph with fewer data points is from simulation with an original Mousetrap module. Naturalizing its links with the circuit of Figure 6(c) produces the lower graph. The increased number of logic gates costs performance. Using the link circuits of Figure 6(b) produces the upper graph, improving the original performance. These three 90nm simulations omit data latches and wire loads.

Did You Know

that a ring of original Mousetrap modules cannot possibly hold an odd number of tokens? The same is true for rings of original Micropipeline and Click modules.

The reason is that all three circuit families *fuse together* a forward request and a reverse acknowledge wire.

- To see why odd initialization is impossible start with an empty ring. During initialization, any change in state of a fused wire changes the state of *two* links. The change will either fill one link and drain the other link, fill both links, or drain both links. Each change keeps the number of full links even, and so the number of full links cannot be odd.
- In contrast, naturalized links can be initialized to full or empty independent of and without changing adjacent links.

This little recognized truth appears clearly in Figure 8

- Although all rings have 24 stages, only the two naturalized Mousetrap graphs have sample points for all occupancies.
- The center graph for original Mousetrap can plot throughput only for even link occupancy, offering fewer sample points.

Naturalized communication
restores the generality
lost to the original circuit families

IV. Naturalized Testing

In the context of this paper "testing" means validating whether or not the fabricated design-on-silicon operates as intended [1]. This includes *structural testing*, by which we mean low-speed testing to uncover fabrication defects, as well as *functional* and *at-speed testing* to uncover incorrect or marginal functionality.

A great deal of the wisdom for testing self-timed circuits comes from testing synchronous circuits. When its clock "ticks" the synchronous circuit acts. It acts by using the present state to compute a next state, which takes over at the next tick. Many synchronous test solutions use some form of *scan test* to control and observe state. They re-use the existing clock to start and stop the test action [13]. This works because each cycle in the design contains a clocked state-holding element. The "tick" governs every synchronous loop.

What makes self-timed circuits "tick" ? [2] One may be tempted to point at local clocks, but these are just by-products. Self-timed circuits act upon the state of their links. Links meet at joints. Each cycle in the design contains a joint. This makes joints the ideal place to start and stop self-timed action.

We emphasize that testing requires access to both *state* and *action*. Unfortunately, the two are often confounded: the action part of a test solution is often integrated into the state part. Once integrated, it becomes much harder to separate them in order to reduce test access costs or to fine-tune or re-use test solutions for debug. With this paper, we call attention to actions, and let them play their own part in the test solution.

In the following sub-sections, we introduce a new circuit element, MrGO, dedicated to actions, which it can safely start, stop, and freeze. MrGO fits into joints. Combined with scan-test based access to naturalized links, joints with MrGO provide a rich environment for test and debug.

A. Takeoff: From Initialization to Self-Timed Operation

Because each naturalized link stores its own full-empty state, links require initialization. Some circuit families, like original GasP, used specialized master-clear circuitry to initialize links with fixed values, typically "empty." Others, like Click, use a scan chain to initialize links with different values.[3]

Some initial link states may evoke instant action from joints. If one permits joints to act during initialization, a joint's action may conflict with initialization. We advocate adding a *go* signal to each and every joint. The *go* signal can be yet another guard term anywhere in the joint's AND-function — see Figure 7. A de-asserted (low) *go* signal makes the joint's fill and drain signals low, thereby freezing the joint. Frozen joints cannot conflict with initialization.

Both the initialization signals and the *go* signal may suffer long and varied delays from their source to remote parts of a large system. Because of differences in these delays, initialization may end at different times in remote parts of the system. Likewise an asserted *go* signal may arrive, unfreeze, and start operations for different parts of the system at different times. A correct start after initialization depends only on avoiding conflict between initialization and operation at every joint.

Initialization may include state-holding elements in the joints, like those used for selective link control. We can deliver initialization signals via a scan chain. A single global *go* signal would suffice to freeze the system for initialization.

[2] Kees van Berkel, thank you for initiating this pun in your ASYNC 1999 Industry Demo presentation "The PCA5007 Pager IC: What Makes it Tick." We have encountered it several times since, but never before used it ourselves.

[3] The Click paper [5] includes a solution for scanning the flipflops that determine the link states, and provides references to related work for scanning other types of state-holding elements used in self-timed circuit designs.

B. Landing: Stopping a Self-Timed Operation in Full flight

Molnar et al. recognized long ago how to stop a self-timed circuit [3]. When a self-timed circuit is told to stop, it must decide cleanly whether to stop at once or to complete a pending or underway action. Because the stop signal is entirely independent of internal signals, a proper stopper must provide for metastability delay. In other words, a proper stopper must contain an *arbiter* or *mutual exclusion element* [7].

Once stopped, it is useful to sense the state of the system. The same scan chain that provides initial values — see Section IV-A — can sense the state of the links and the joints. The proper stopper can be added anywhere in the joint.

C. MrGO

We have found it convenient to combine the *go* signal and the arbitrated stop in one circuit: MrGO, pronounced "Mister GO" — see Figure 9(top). When appended to the AND-functions of the joints, as in Figure 9(bottom), it serves as proper starter for happy takeoffs and as proper stopper for happy landings. MrGO also helps us test the circuit, as we will show next.

For test control, it is essential that MrGO be placed inside the joint-link-joint feedback loops. This ensures that any arbitration contest between *go* and local self-timed signals will resolve and end with the *go* signal taking control of the arbiter. To start and stop each and every joint individually, each MrGO gets a separate *go* signal from a scan chain.

D. Testing with MrGO: Single- and Multi-Step Operations

Selectively asserted *go* signals provide a wide variety of test options. We list some below. We can make each test insensitive to delay variation in different *go* signals.

1) One-shot Test of a Selected Joint:
Initialization sets the link states and internal states and data for the joint to be tested. With all other joints frozen, permitting the selected joint to go lets it take at most one action.[4] After re-freezing the selected joint, examination of its links and internal state reveals if it took the expected action.

2) Following a Thread of Action:
A sequence of one-shot tests can follow a data item along a pipeline. Each one-shot test advances the data item to joints that might act were they not frozen. The next step freezes the joint that previously acted and then permits the next joint to go. Allowing joints to go only one at a time makes it possible to track the flow of data items through a system. See Figure 13.

3) Breakpoint:
Testing a rarely used part of a system, such as memory error correction, is possible by freezing one or more joints there. Full-speed action of the rest of the system will stop at calls for action of a frozen joint.

4) Testing a Single Data Item At Speed:
This test setup leaves several adjacent joints unfrozen to permit a data item to pass at speed through them. A frozen joint upstream of this test section blocks entry of test data input. A frozen joint downstream prevents escape of test data output. Unfreezing the upstream frozen joint releases the test data item to flow through the test section at speed. See Figure 10.

5) Testing a Burst of Data Items At Speed:
Just as a single data item can flow through a test section, so can a burst of data items. The burst of data items queues up behind the upstream frozen joint much like water behind a dam. Unfreezing the joint releases the burst. There must be data storage for the entire burst in the release and capture sections ahead of and behind the test section. See Figure 12.

6) Testing the Flow of Bubbles At Speed:
Canopy graphs (Figure 8) teach us that data flowing forward through a pipeline tend to move at a different speed than bubbles flowing backward. We have learned through painful experience that testing the flow of bubbles through a congested pipeline is as important as testing the flow of data through an empty pipeline. A test section initialized with full rather than empty links reveals its response to bubbles, allowing detection of faulty behaviors often overlooked. See Figure 11.

Test options 1–3 are useful for structural testing, for instance for stuck-at faults. Options 3–6 are useful for testing delay faults and marginal functionality. For debug, all options matter.

[4]For example, the joint in Figure 9(bottom) will either do nothing or generate a high pulse on its fill and drain signals, changing both links.

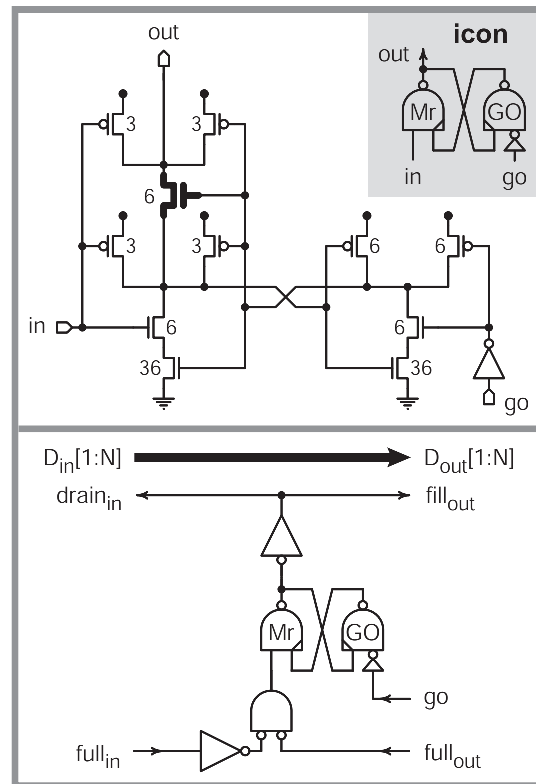

Figure 9 MrGO with its icon inset in the grey area (top), and a joint with MrGO (bottom). The bold central transistor in MrGO delays active-low grant signal, *out*, by conducting only after metastability ends. Transistor sizing reduces the logical effort from *in* to *out*. Split pull-up transistors in the left NAND gate avoid a floating *out* signal. Selective metastability-protected freezing (*go* is low) and unfreezing (*go* is high) of joints provides for testing. MrGO is inspired by the HOLD design-for-test solution [6], proper stopper [3], and Seitz' mutual exclusion element [7].

test command							counter stage							count
init	[1]	-	[2]	-	[3]	-	[4]	-	[5]	-	[6]	-	[7]	0
tunnel	[1]	$\xrightarrow{A}$	[2]	-	3	-	4	-	5	-	[6]	-	[7]	0
run	[1]	-	2	$\xrightarrow{A}$	3	-	4	-	5	-	[6]	-	[7]	0
	[1]	-	2	-	3	$\xrightarrow{A}$	4	-	5	-	[6]	-	[7]	0
$\vdots$	[1]	-	2	-	3	-	4	$\xrightarrow{A}$	5	-	[6]	-	[7]	1
done	[1]	-	2	-	3	-	4	-	5	$\xrightarrow{A}$	[6]	-	[7]	1

Figure 10 (testing the counter in pipeline stage 4 with a single data item, at speed)
The top row (*init*) shows a pipeline segment with seven joints, 1–7, and a counter attached to joint 4. Initially all seven joints are frozen, illustrated by the square brackets "[" and "]" around each joint, all links between them are empty, illustrated by the simple dash "-" for each link, and the counter value, *count*, is *0*. Next, as shown in row 2, we prepare a test section (*tunnel*) to test the counter at speed by permitting joints 3, 4 and 5 to go when possible, illustrated by the absent brackets. In addition, we fill the link between joints 1 and 2 with one data item, *A*, illustrated by the labeled arrow. None of the joints can act yet, but as soon as we permit joint 2 to go, as shown in row 3 (*run*), data item *A* moves to the right as far and as fast as it can, incrementing the counter.

test command							counter stage							count
init	[1]	$\xrightarrow{A}$	[2]	$\xrightarrow{B}$	[3]	$\xrightarrow{C}$	[4]	$\xrightarrow{D}$	[5]	$\xrightarrow{E}$	[6]	$\xrightarrow{F}$	[7]	0
tunnel	[1]	$\xrightarrow{A}$	[2]	$\xrightarrow{B}$	3	$\xrightarrow{C}$	4	$\xrightarrow{D}$	5	$\xrightarrow{E}$	[6]	-	[7]	0
run	[1]	$\xrightarrow{A}$	[2]	$\xrightarrow{B}$	3	$\xrightarrow{C}$	4	$\xrightarrow{D}$	5	-	6	$\xrightarrow{E}$	[7]	0
	[1]	$\xrightarrow{A}$	[2]	$\xrightarrow{B}$	3	$\xrightarrow{C}$	4	-	5	$\xrightarrow{D}$	6	$\xrightarrow{E}$	[7]	0
$\vdots$	[1]	$\xrightarrow{A}$	[2]	$\xrightarrow{B}$	3	-	4	$\xrightarrow{C}$	5	$\xrightarrow{D}$	6	$\xrightarrow{E}$	[7]	1
done	[1]	$\xrightarrow{A}$	[2]	-	3	$\xrightarrow{B}$	4	$\xrightarrow{C}$	5	$\xrightarrow{D}$	6	$\xrightarrow{E}$	[7]	1

Figure 11 (testing the counter in pipeline stage 4 with a single bubble, at speed)
This test complements the one in Figure 10. The top row (*init*) shows all seven joints frozen, illustrated by the square brackets around them, all links between them full, indicated by the labeled arrows, and a counter value of *0*. Next, as shown in row 2, we prepare an at-speed test section (*tunnel*) through joints 3, 4 and 5, as illustrated by the absent brackets. In addition, we empty the link between joints 6 and 7, introducing one bubble, illustrated as "-." None of the joints can act yet, but as soon as we permit joint 6 to go, in row 3 (*run*), the bubble moves to the left as far and as fast as it can. The counter increments as token *C* moves past.

test command									weak stage						
init	[1]	$\xrightarrow{A}$	[2]	…	[6]	$\xrightarrow{F}$	[7]	-	[8]	-	[9]	…	[13]	-	[14]
tunnel	[1]	$\xrightarrow{A}$	2	…	6	$\xrightarrow{F}$	[7]	-	8	-	9	…	13	-	[14]
run at lower V_{DD}	[1]	$\xrightarrow{A}$	2	…	6	$\xrightarrow{F}$	7	-	8	-	9	…	13	-	[14]
$\vdots$															
done	[1]	-	2	…	6	-	7	-	8	$\xrightarrow{A}$	9	…	13	$\xrightarrow{F}$	[14]

Figure 12 (testing the marginal latch in pipeline stage 8 with a burst of data items, at speed)
Using the same notation as in Figure 10, the top row (*init*) shows a pipeline segment with frozen joints and six full links, followed by a frozen *weak stage* — a joint with a marginal latch — followed by a pipeline segment with frozen joints and six empty links. The data items for the full links are shifted into place as explained in Figure 13. Next, as shown in row 2, we prepare an at-speed test section (*tunnel*) through joints 8 to 13, as illustrated by the absent brackets. None of the joints can act until we permit joint 7 to go. Before giving permission, in row 3 (*run*), we reduce the supply voltage to aggravate the error condition of the marginal latch. The resulting behavior is captured in the pipeline segment after the *weak stage*. Various data patterns of successive bits such as *101010*, *110110*, *001001* exercise the weak latch. Competing patterns for adjacent bits can check for sensitivity to crosstalk.

test command															weak stage
init	[1]	-	[2]	-	[3]	-	[4]	-	[5]	-	[6]	-	[7]	-	[8]
go	1	$\xrightarrow{F}$	[2]	-	[3]	-	[4]	-	[5]	-	[6]	-	[7]	-	[8]
nogo	[1]	$\xrightarrow{F}$	[2]	-	[3]	-	[4]	-	[5]	-	[6]	-	[7]	-	[8]
go	[1]	-	2	$\xrightarrow{F}$	[3]	-	[4]	-	[5]	-	[6]	-	[7]	-	[8]
nogo	[1]	-	[2]	$\xrightarrow{F}$	[3]	-	[4]	-	[5]	-	[6]	-	[7]	-	[8]
$\vdots$															
nogo	[1]	-	[2]	-	[3]	-	[4]	-	[5]	-	[6]	$\xrightarrow{F}$	[7]	-	[8]

Figure 13 (shifting the data items into place for the marginal latch test of Figure 12)
Like non-overlapping clocks *go* and *nogo* commands can shift a single data item or many data items at once through a pipeline. Rows 1–13 above, shift exactly one data item, *F*, into place for row 1 of Figure 12. Similar steps place data items *E* through *A*. The low activity factor of such a single-shift approach makes it possible to shift data reliably to and from the marginal latch.

V. Silicon Test and Debug Examples

We have two working chip experiments, *Weaver* and *Anvil*, both built in 40nm TSMC CMOS. Both experiments use rings of normally-opaque GasP pipelines with naturalized communication to recirculate data at high speed. Anvil has a MrGO circuit in each and every GasP stage, i.e. joint, to provide *go* control, including arbitrated stop. Weaver has *go* control in each and every GasP stage and arbitrated stop capability in nearly all stages. Although the intended purposes of these two silicon experiments are beyond the scope of this paper, the few examples in this section show how naturalized testing has provided assurance of their correct operation. Weaver and Anvil each have an IEEE standard JTAG test access port and scan interface [4] to provide a low speed interface to their high-speed operation. Software in a control computer sets up and runs test commands, and evaluates the results of each test.

Each of several re-circulating rings in Weaver and Anvil has a binary counter attached to one of its stages. The counter is supposed to increment each time a data item passes by. Reading counts via the JTAG interface before and after a full-speed run of known duration allows test software to compute throughput and to make canopy graphs of each ring. The Weaver has measured throughput of about 6 Giga data items per second. Before using a counter, we test its correct operation using tests through pipeline segments around the counter stage — like the two at-speed tests for a single data item and a single bubble passing the counter in Figures 10–11. Correct counter operation is essential to many tests and measurements.

Note that although it takes hours to write the software to test correct operation of a counter, and milliseconds for the test computer to set up suitable initial conditions, the actual tests in Figures 10 and 11 run to completion in half a nanosecond.

Anvil includes latches of many different designs. One particular latch gave erratic results at reduced power supply voltage. Exploring the details of its behavior required testing the marginal latch with a variety of data patterns run at speed with reduced supply voltage, V_{DD}. Figure 12 shows how selective *go* control made this possible.

Anvil limits the area cost of its scan chain by limiting the number of places in a relatively long pipeline where the scan chain can insert or retrieve data values. As a result, Anvil allows data entry only dozens of pipeline stages ahead of the marginal latch. Moreover, because the noise of full-speed operation might cause errors, slow and careful delivery of the test patterns seemed essential. The method described in Figure 13 provides slow but accurate delivery of suitable data input patterns shown in the first row of Figure 12 and slow but accurate retrieval of data results shown in the last row.

Weaver has some joints that steer data items to alternate links. To detect a data item that might stray outside its planned path, test software freezes all joints outside that path. Each such frozen joint acts as a *breakpoint* — stray data items fill links that terminate at such frozen joints. An overview of scanned-out full links quickly identifies not only that there are stray data items, but also whither they have strayed.

VI. Conclusion

This paper is built around a novel point of view. We differentiate *links* from *joints* and *actions* from *states*, empowering each to play its natural role during system design and test.

Differentiating links from joints is a simple idea of great power because it offers a higher level of abstraction for each. The simple interface between links and joints of Figure 5 and the resulting unification of four bundled-data two-phase circuit families attest to the impact of the new abstractions. Such abstractions and the unification of families simplify computer aided design, and herald circuit improvements that extend the geographic reach and reduce the energy consumption of links.

Differentiating actions from states is a simple idea of great power because it clarifies how self-timed systems work. Unlike actions in synchronous systems that occur simultaneously in response to an external clock, the actions of self-timed systems are spontaneous, self-generated, and widely distributed in both space and time. Separate action control with MrGO of Figure 9, combined with traditional scan access to state, enables single-step and local at-speed operations essential to silicon test and debug. Although every action changes local state, we can test that those actions conform to expectation.

Acknowledgment

We thank corporate and private sponsors of the Asynchronous Research Center. Jon Lexau of Oracle smoothed the myriad details required to plan, make and mount Weaver and Anvil. We thank Bert Sutherland, Bob Sproull, and Prof. Xiaoyu Song for encouragement and discussions. We thank Prof. Rob Daasch for rolling up his sleeves and testing alongside us. Last but not least, we thank Jens Sparsø for shepherding this paper.

References

[1] Michael Bushnell and Vishwani Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Springer, 2005.

[2] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Long-Range GasP with Charge Relaxation. In *Asynchronous Circuits and Systems (ASYNC)*, pages 185–195, 2010.

[3] Charles Molnar, Ian Jones, William Coates, and Jon Lexau. A FIFO Ring Performance Experiment. In *Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 279–298, 1997.

[4] The Institute of Electrical and Electronics Engineers. IEEE Standard Test Access Port and Boundary-Scan Architecture (1149.1), 2001.

[5] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. In *Asynchronous Circuits and Systems (ASYNC)*, pages 3–14, 2010.

[6] Marly Roncken. Defect-Oriented Testability for Asynchronous ICs. *Proceedings of the IEEE*, 87(2):363–375, 1999.

[7] Charles Seitz. Chapter 7: System Timing. In *C. Mead and L. Conway: Introduction to VLSI Systems*, pages 218–262. Addison-Wesley, 1980.

[8] Montek Singh and Steve Nowick. Mousetrap: High-speed Transition-Signaling Asynchronous Pipelines. *Transactions on VLSI Sytems*, 15(6):684–698, 2007.

[9] Jens Sparsø and Steve Furber (Eds.). *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[10] Ivan Sutherland. GasP Circuits that Work. ECE 507 research seminar, Fall 2010. Asynchronous Research Center, Portland State University. Download from http://arc.cecs.pdx.edu/fall10.

[11] Ivan Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.

[12] Ivan Sutherland and Scott Fairbanks. GasP: A Minimal FIFO Control. In *Asynchronous Circuits and Systems (ASYNC)*, pages 46–53, 2001.

[13] Neil Weste and David Money Harris. *CMOS VLSI Design — A Circuits and Systems Perspective (Fourth Edition)*. Addison Wesley, 2011.