Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

Winter 1-17-2018

Silicon Compilation and Test for Dataflow Implementations in GasP and Click

Swetha Mettala Gilla
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the Digital Circuits Commons, and the Electrical and Computer Engineering Commons

Let us know how access to this document benefits you.

Recommended Citation
Mettala Gilla, Swetha, "Silicon Compilation and Test for Dataflow Implementations in GasP and Click" (2018). *Dissertations and Theses.* Paper 4237.
https://doi.org/10.15760/etd.6121

This Dissertation is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Silicon Compilation and Test for Dataflow Implementations in GasP and Click

by

Swetha Mettala Gilla

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
in
Electrical and Computer Engineering

Dissertation Committee:
Xiaoyu Song, Chair
Douglas V. Hall
Robert W. Daasch
Ivan E. Sutherland
Mark P. Jones
Marly Roncken (non-voting member)

Portland State University
2018

© 2018 Swetha Mettala Gilla

ABSTRACT

Many modern computer systems are distributed over space. Well-known examples are the Internet of Things and IBM's TrueNorth for deep learning applications. At the Asynchronous Research Center (ARC) at Portland State University we build distributed hardware systems using self-timed computation and delay-insensitive communication. Where appropriate, self-timed hardware operations can reduce average and peak power, energy, latency, and electro-magnetic interference. Alternatively, self-timed operations can increase throughput, tolerance to delay variations, scalability, and manufacturability.

The design of complex hardware systems requires design automation and support for test, debug, and product characterization.

My PhD thesis focuses on design compilation and test support for dataflow applications. Both parts are necessary to go from self-timed circuits to large-scale hardware systems.

As part of my research in design compilation, I have extended the ARCwelder compiler designed by Willem Mallon (previously with NXP and Philips Handshake Solutions). The resulting ARCwelder compiler can support multiple self-timed circuit families.

The key to testing distributed systems, including self-timed systems, is to identify the actions in the systems. In distributed systems there is no such thing as a global action. To test, debug, characterize, and even initialize distributed

systems, it is necessary to control the local actions - individually! The designs that we develop at the ARC separate the actions from the states *ab initio*.

As part of my research in test and debug, I implemented a special circuit to control actions, called MrGO. I also implemented a scan and JTAG test interface to take control over each individual and local action, each individual and local communication state, and any subset of data-related state elements one might wish to control or observe. My test implementations have been built into two silicon test experiments, called Weaver and Anvil, and were used successfully for testing, debug, and performance characterizations.

To my advisor, Marly, and my daughter, Swara.

Acknowledgments

Firstly, I want to thank my advisor, Drs. Marly Roncken, for her guidance during my research and study at Portland State University (PSU). She is not only my advisor, she is also a mentor and my role model. Her push and consistent motivation helped cross hurdles and gave me the courage and patience to finish my thesis.

I was delighted to interact with Dr. Ivan Sutherland. It was a great opportunity to work with him on arbiter circuits. His insight in asynchronous circuit design is amazing. He explains anything that looks like "eschew obfuscation" into something as obvious as "keep it simple" [1]. To me, he defines what a world-class researcher and teacher are about.

I am much obliged to Professor Xiaoyu Song, for agreeing to be my thesis advisor. One simply could not wish for a more friendly and understanding advisor. Professor Douglas V. Hall, Professor Robert Daasch and Professor Mark Jones deserve special thanks as my dissertation committee members. I sincerely appreciate their participation and technical feedback. I thank Willem Mallon for sharing his knowledge on compilers. Willem is the greatest programmer I have ever met, as well as a selfless and kind human being.

Special thanks go to my colleagues Chris Cowan, Navaneeth Jamadagni, Hoon

[1] The text "eschew obfuscation" is on one of his sweatshirts.

Park and Chris Chen for the many technical and non-technical discussions and for making this research fun.

I am indebted to my parents and my in-laws for their care and love. I thank you for believing in me and for supporting my study and career decisions.

I owe thanks to a very special person, my husband Ravi, for his continued love and support. He deserves the same amount of credit as I do for this work. I thank my baby, my little girl Swara, for abiding my absence. My heartfelt thanks go to my mother-in-law for taking my place in mothering Swara during the last phase of my PhD research.

Table of Contents

Abstract **i**

Dedication **iii**

Acknowledgments **iv**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1. Research Objective and Approach 2

1.2. Research Summary 3

1.3. My Contributions 6

1.4. Thesis Organization 7

2 Telescope GasP (TGasP) **10**

2.1. Why Telescope GasP? 11

2.2. Traditional versus Telescope GasP 14

2.3. Cost of and Alternatives to Telescope GasP 18

2.4. Designing Telescope GasP Modules 21

2.5. Latency Reduction in the Reverse Direction 26

2.6. Summary and Conclusions 29

2.7. My Contributions for Chapter 2 31

3 Silicon Compilation for Click and GasP 32

3.1. Existing ARCwelder Compiler 34

3.1.1. Graphical User Interface 36

3.1.2. Parser 36

3.1.3. Static Validation 36

3.1.4. Dynamic Validation 36

3.2. ARCwelder Compiled Circuits: Existing versus New 37

3.3. ARCwelder for Click and (T)GasP 42

3.3.1. Graphical User Interface Changes 42

3.3.2. Parser Changes 44

3.3.3. Static Validation Changes 45

3.3.4. Dynamic Validation Changes 50

3.4. ARCwelder Click versus (T)GasP Experiments 52

3.5. Summary and Conclusion 58

3.6. My Contributions for Chapter 3 59

4 Naturalized Communication 61

4.1. Systems with Handshake Interfaces 63

4.2. Systems with Full-Empty Interfaces 70

4.2.1. Naturalized Links and Joints 73

4.3. Impact of Naturalization on System Design 76

4.4. Summary and Conclusion 78

4.5. My Contributions for Chapter 4 79

5 Naturalized Testing 80

5.1. Testing: Where we Came from, Are, and Go 80

5.2. Naturalized Testing 84

5.2.1. Takeoff: From Initialization to Self-Timed Operation 85
5.2.2. Landing: Stopping a Self-Timed Operation in Full Flight . 85
5.2.3. MrGO 86
5.3. Testing with MrGO 86
5.3.1. One-shot Test of a Selected Joint 88
5.3.2. Following a Thread of Actions 88
5.3.3. Breakpoints 88
5.3.4. Testing a Single Data Item at Speed 91
5.3.5. Testing a Burst of Data Items at Speed 91
5.3.6. Testing the Reverse Flow of “Bubbles” at Speed 91
5.3.7. Stuck-at Fault Coverage 95
5.4. Summary and Conclusion 96
5.5. My Contributions for Chapter 5 97

6 Hierarchical Design for Test and Debug **99**
6.1. Module level Test and Debug 102
6.1.1. Proper Start Control: Make *go* High 106
6.1.2. Proper Stop Control: Make *go* Low 107
6.1.3. Full-empty State Control and Observation 108
6.1.4. Data Control and Observation 109
6.2. Design Level Test and debug 109
6.2.1. Scan Shift 116
6.2.2. Non-Destructive Read of Circuit Signals 119
6.2.3. Non-Destructive Write of Circuit Signals 120
6.2.4. Initialization at Power-Up 120
6.3. Chip Level Test and debug 121
6.4. Summary and Conclusion 127
6.5. My Contributions for Chapter 6 129

7 Arbitration 130

7.1. A Tale of Two Arbiters 131

7.1.1. How the Arbiter Circuits Work 134

7.1.2. Least Uncontested Delay 135

7.1.3. Our Simulation and Analysis 136

7.2. Mathematical Analysis 136

7.2.1. Parameters: Load, Strength, Delay 137

7.2.2. Overall Delay Analysis 141

7.2.3. ARC Arbiter Analysis 142

7.2.4. Sparsø-Furber (SF) Arbiter Analysis 144

7.2.5. Mathematical Delay Analysis: Summary and Comparison 146

7.3. SPICE Delay Simulations 149

7.4. Comparison of Analyzed versus Simulated Results 152

7.5. Sizing Validation from a Design Perspective 153

7.6. Summary and Conclusion 158

7.7. My Contributions for Chapter 7 159

8 Conclusion and Future work 161

References 164

Appendix A Telescope GasP: Storage and Broadcast 173

Appendix B Telescope GasP: Narrowcast 206

Appendix C Telescope GasP: Repeat 262

Appendix D Naturalized Communication and Testing 281

Appendix E Test Setup: Scan Chain Organization 290

Appendix F **Test Setup: JTAG Controller** **308**

Appendix G **Arbiter Design: Reduced Input Load** **350**

Appendix H **Arbiter Design: Noise Robustness** **372**

Appendix I **Bonus: PhD Defense Presentation** **384**

List of Tables

Table 3.4.1: Path delay information for (T)GasP and Click modules 54
Table 3.4.2: GasP timing information for Fibonacci number generator . . . 56
Table 3.4.3: Click timing information for Fibonacci number generator. . . . 57

Table 7.2.1: Transistor equations for input capacitance versus strength. 139
Table 7.2.2: Equations for best transistor strengths for the ARC arbiter . 144
Table 7.2.3: Equations for best transistor strengths for the SF arbiter. . . 146
Table 7.2.4: Mathematically best strengths and grant delays 148
Table 7.3.1: SPICE-simulated best strengths and grant delays 152
Table 7.4.1: Comparison of analyzed versus simulated grant delays . . . 153

List of Figures

Figure 2.1.1: An example of a FIFO using TGasP modules 13
Figure 2.2.1: GasP Store circuit implementation 15
Figure 2.2.2: GasP Store SPICE waveforms 16
Figure 2.2.3: TGasP Amplify circuit implementation 17
Figure 2.2.4: TGasP Amplify SPICE waveforms 17
Figure 2.4.1: Second alternative TGasP Amplify circuit implementation . 22
Figure 2.4.2: Third alternative TGasP Amplify circuit implementation 23
Figure 2.4.3: Second alternative TGasP Amplify SPICE waveforms 24
Figure 2.4.4: Delay circuits to increase timing margins 25
Figure 2.5.1: Fast reset configuration for TGasP 27
Figure 2.5.2: Fast reset SPICE waveforms for TGasP 28

Figure 3.1.1: ARCwelder organization 35
Figure 3.2.1: Store circuit implementations in Click and GasP 37
Figure 3.2.2: Fork circuit implementation in Click and GasP 40
Figure 3.3.1: GUI design for Fibonacci number generator 43
Figure 3.4.1: GUI alternative design for Fibonacci number generator 53

Figure 4.0.1: A self-timed dataflow system with links and joints 62
Figure 4.1.1: Pictorial representation of a self-timed FIFO 63
Figure 4.1.2: Full-empty representations for two-phase handshakes 66

Figure 4.1.3: Original self-timed FIFO circuit in GasP and Click 66
Figure 4.1.4: Pictorial view of a FIFO with GasP and Click modules. 68
Figure 4.1.5: Mixed systems with handshake interfaces need translators 69
Figure 4.2.1: GasP circuit with full-empty interface 71
Figure 4.2.2: Click circuit with full-empty interface 72
Figure 4.2.3: Pictorial view of a mixed FIFO with full-empty interfaces . . . 74
Figure 4.2.4: Design sketch of a naturalized broadcast joint 75
Figure 4.3.1: Canopy graph for original and naturalized designs 77

Figure 5.1.1: Where we came from: synchronous state-based test view . 81
Figure 5.1.2: Where we are: state-based test view for self-timed 82
Figure 5.1.3: Where we go: state-action based test view for self-timed . . 83
Figure 5.2.1: GasP circuit with MrGO 87
Figure 5.3.1: Representation of a one-shot test of a selected joint 89
Figure 5.3.2: A series of one-shot tests to prime a marginal latch test . . . 89
Figure 5.3.3: Representation of a breakpoint test 90
Figure 5.3.4: Pictorial representation of an at-speed test for a data item . 92
Figure 5.3.5: Textual representation of an at-speed test for a data item . . 92
Figure 5.3.6: Pictorial representation of an at-speed data burst 93
Figure 5.3.7: At-speed test of a marginal latch using a data burst 93
Figure 5.3.8: Pictorial representation of an at-speed test for bubbles 94
Figure 5.3.9: Textual representation of an at-speed test for bubbles 94
Figure 5.3.10: Test outline for exhaustive testing of stuck-at faults 95
Figure 5.5.1: Poster, illustrating the test solution with MrGO 98

Figure 6.0.1: Block diagram of the hierarchical design for test approach 100
Figure 6.1.1: GasP FIFO with links and joints and test signals 103
Figure 6.1.2: Transistor level schematic for a 1-bit datapath latch 105

Figure 6.2.1: Pictorial representation of a GasP FIFO with scan chains 110
Figure 6.2.2: Scan operations for a one-shot FIFO test. 113
Figure 6.2.2: Scan operations for a one-shot FIFO test (continued) 114
Figure 6.2.3: Scan segment implementations for GasP 117
Figure 6.2.4: Scan segment implementations for GasP (continued) 118
Figure 6.3.1: Block diagram of the JTAG controller 122
Figure 6.3.2: Scan Reset Mux to support initialization at power-up. 126

Figure 7.1.1: Transistor level schematic of the ARC arbiter 133
Figure 7.1.2: Transistor-level schematic of Sparsø-Furber (SF) arbiter . 134
Figure 7.3.1: SPICE-simulated delay graphs for the two arbiters. 150
Figure 7.5.1: ARC arbiter with optimized strengths 155
Figure 7.5.2: SF arbiter with optimized strengths 155
Figure 7.5.3: Optimally sized arbiter graphs: grant, reset, logical effort . 157

1

Introduction

Many modern computer systems are distributed over space. Well-known examples are Internet of Things and IBM's TrueNorth for Deep Learning applications. In clocked synchronous systems, large modern chips require multiple clock domains. High clock frequencies are possible only in limited-area clock domains [21]. Data synchronization between the clock domains is becoming increasingly onerous. At the Asynchronous Research Center (ARC) at Portland State University we build distributed hardware systems using self-timed computation and delay-insensitive communication. Self-timed design offers a way to coordinate actions between multiple time domains without using synchronizers. The flexibility provided by self-timed coordination makes it easier to change designs so they fit in a power or speed or time-to-market schedule. Self-timed design may be inevitable [54].

To make the design of self-timed systems practical, we must have automated design tools [7]. Current access to design automation tools for self-timed systems is very restrictive. Some of the best known tools are proprietary [5, 3]. Also, usage of the self-timed design paradigm is inhibited by difficulties related to testing [62]. Testing self-timed systems often requires dealing with combinational loops and distributed actions – both of which are handled poorly by

standard tools for synchronous designs [59].

My PhD thesis focuses on both design compilation and test support for self-timed designs, particularly self-timed designs for dataflow applications. Both parts are necessary to go from self-timed circuits to large-scale hardware systems. The key insights resulting from this research have already led to generalizations and similar insights for other self-timed design approaches and tools [46].

1.1 Research Objective and Approach

Objective

The primary objective of this research is to provide generic design compilation and test support for self-timed circuits.

Approach

My research doesn't start from scratch. I use existing self-timed design families called GasP [55] and Click [43]. I use an existing design compiler tool called ARCwelder. I use existing scan test support approaches developed for testing synchronous designs and adapted for testing self-timed designs. My motivation for reusing these as my starting points is as follows.

1. Click and GasP are excellent starting points for circuit design, because they are the two extremes of so-called bundled-data self-timed circuit families – with GasP the most asynchronous and Click the most synchronous self-timed circuit style.

2. ARCwelder is an excellent starting point for design automation, because it compiles dataflow designs already into Click implementations. ARCwelder

was built by Willem Mallon, a compiler and self-timed circuit expert, who based this compiler design on his prior knowledge and expertise as a core team member of Philips Handshake Solutions [43, 42].

3. Scan is an excellent starting point for test and test automation, because as ARC researchers we have access to scan test software and hardware equipment used by Sun-Oracle Laboratories to test GasP integrated circuits.

None of these three categories of starting points provide a generic design and test solution by themselves. But, together they can teach us where design and test automation procedures for Click and GasP differ and how we might introduce a new point of view to emphasize their same-ness in terms of design and test automation, while maintaining their uniqueness in terms of low power, high speed and latency tolerance.

1.2 Research Summary

To compile a given dataflow design into self-timed circuits that are implemented using a GasP or a Click circuit design style, we partition the design into building blocks and provide a GasP or Click implementation for each building block.

We have building blocks for buffering, data-driven selection, branch and merge operations, and so on. In GasP, all existing building blocks were fully pipelined, storing their own data. In Click, however, most of the building blocks that Willem Mallon had built into the ARCwelder compiler were pipelined only in their topological ordering, and few stored data. Willem separated building blocks into blocks for data storage and blocks for flow control. Willem called the blocks that store data "*Storage* modules." He called flow control blocks by their function,

e.g. *Merge, Join, Fork*, and *Distribute*. Except for Storage modules, all other modules in ARCwelder refrain from storing data.

To compile the same ARCwelder dataflow designs in both Click and GasP, I developed a new collection of GasP modules. The new GasP modules implement flow control modules without storage for which Willem had a Click implementation in ARCwelder. I named this new collection "*Telescope GasP*" — *TGasP* in short — because the communication behavior of the new modules resembles the folding and un-folding of a telescope.

In theory, all that the ARCwelder compiler must now support is a one-to-one mapping of dataflow building blocks, connected into a directed graph. Each building block is mapped to either a Click module implementation or a GasP or TGasP module implementation. In practice, this turned out to be more difficult than we had anticipated. Differences that we had expected to be minor, such as circuit initialization, and other differences that we had expected to be modular, such as communication protocol checks, incurred changes at multiple levels in the code and more code duplications than we felt were reasonable for a unified compiler solution. The documentation in this thesis shows where ARCwelder can be made more robust and more generic in supporting Click and (T)GasP designs.

The unanticipated difficulties in reusing ARCwelder to compile dataflow designs into either Click or (T)GasP circuits served as an eye opener for how we, at our Asynchronous Research Center (ARC) and in the asynchronous research community at large, have created a "Tower of Babel" by using different communication protocols, different initialization schemes, and different design and test approaches.

As a result of my compiler study, we at the ARC started emphasizing the similarities between the various self-timed circuit families. In doing so, we created a new point of view that applies to the design and test of self-timed circuits in general [46]. This new point of view, will, I believe, also lead to a unified compiler solution for mapping dataflow designs into self-timed circuits of any style or mix of styles.

For the rest of my thesis research, I have worked out various parts related to the test solution presented in the 2015 publication by Roncken et al. [46]. The key to testing distributed systems, including self-timed systems, is to identify the actions in the system. In a distributed system there is no such thing as a global action. To test, debug, characterize, and even initialize a distributed system, it is necessary to control the local actions individually! The designs that we develop at the ARC separate the actions from the states *ab initio*.

As part of my research in test and debug, I implemented a special circuit to control actions, called MrGO. I also developed a scan and JTAG test interface to take control over each individual and local action, each individual and local communication state, and any subset of data related state elements one might wish to control or observe. My test implementations keep the system's speed and power unchanged. They have been built into two silicon test experiments, called Weaver and Anvil, and were used successfully for test, debug, and performance characterizations.

The special circuit for action control, MrGO, contains an arbiter. The presence of MrGO in each and every action has increased the number of arbiters in our designs dramatically. Where before we used arbiters only to solve access to shared resources, we now use arbiters in each and every basic computation cycle. Every self-timed action now has an arbitrated MrGO circuit. To develop a

good MrGO implementation it is essential to design a fast and efficient arbiter. I have done extensive research on arbiter designs. I improved the existing ARC arbiter, making it efficient and robust. I also analyzed and compared the ARC arbiter design that we use as the basis for MrGO to the arbiter design used by the majority of self-timed circuit designs elsewhere. My analysis gives great insight into the role of the various transistors in both arbiter designs and how to size them for optimum performance.

1.3 My Contributions

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

- The design and implementation of the Telescope GasP or TGasP family presented in Chapter 2 of this thesis, including the broadcast, narrowcast and repeat modules in Appendices A, B, and C.
- The development of a working scan design and JTAG test interface for GasP presented in Chapter 6 and Appendices E and F.
- The implementation of MrGO – the special circuit that allows us to control actions for initialization, test, debug, and characterization – and more recently mixed synchronous-asynchronous circuit operation [45].
- The development of a series of arbiter designs with lower input load and noise sensitivity, presented in Appendices G and H.
- The delay analysis and optimization in Chapter 7 of the ARC arbiter as used in MrGO, and of an alternative arbiter commonly used outside the ARC and referenced in the Sparsø-Furber book [50].

1.4 Thesis Organization

The rest of this thesis is organized as follows.

- Chapter 2 introduces Telescope GasP or TGasP. TGasP modules supplement the already existing or "traditional" GasP modules. Together, GasP and TGasP form a complete backend for the existing ARCwelder compiler developed by Willem Mallon. This backend is complete in the sense that it serves as a replacement for the existing backend in Click. All previous dataflow designs in ARCwelder can be mapped to Click or to circuit implementations with GasP and TGasP modules. I discuss the design differences between Telescope GasP and traditional GasP. I also show various ways to design TGasP circuits and their pros and cons. A representative set of TGasP modules with correctly sized designs and SPICE simulations can be found in the ARC reports shown in Appendices A, B, and C.

- Chapter 3 discusses our self-timed dataflow design and compilation flow and the ARCwelder compiler. I explain key differences between the Click and GasP circuit families. I explain the various steps involved in extending the ARCwelder compiler to compile dataflow designs into both Click and GasP circuit implementations.

  Note
  The ARCwelder compiler research and experimental results from Chapter 3 have changed the point of view on self-timed design and test at the ARC. The new design point of view is included in my thesis in Chapter 4. The new test point of view is included in my thesis in Chapter 5.

- Chapter 4 introduces the ARC's new point of view on self-timed design [46].

This new design point of view applies to all self-timed circuit families and makes it possible to mix, match, reuse, and exchange designs from different circuit families without further translation. The new approach, dubbed "Naturalized Communication," makes computation as important as communication from the lowest levels of design and up. As a result, the specifics of the various communication protocols and data storage solutions can be hidden inside the design modules, and modules can interact using common, generic interfaces.

- Chapter 5 introduces the ARC's new point of view on initialization and testing of self-timed designs [46]. This new point of view applies to all self-timed circuit families and distributed systems. The new approach, dubbed "Naturalized Testing," makes actions as important as states, from the lowest levels of design-for-test and up. In addition to scan design-for-test circuitry to manage state, we add *go* signals and MrGO circuitry to manage actions. Chapter 5 describes various test experiments using MrGO and scan.

- Chapter 6 gives an overview of hierarchical design-for-test implementations from standard external test interfaces to the specific management of (1) distributed states via traditional scan test circuits and (2) distributed actions via the new MrGO circuitry. Supporting documentation for Chapter 6 with specific implementation details can be found in Appendices E and F.

- Chapter 7 focuses on arbiter designs and compares the ARC arbiter and the Sparsø-Furber arbiter [50]. Supporting documentation for Chapter 7 can be found in Appendices G and H.

- Appendices A, B, and C provide additional details to Chapter 2, Appendix

E and Appendix F provide additional details to Chapters 5–6, and Appendices G and H provide additional details to Chapter 7.

- Appendix D gives the IEEE publication "Naturalized Communication and Testing," published at ASYNC 2015, to which my PhD thesis was instrumental, and which — vice versa — was instrumental to this thesis.

- Appendix I contains the slides of my PhD dissertation defense, presented on the 3$^{\mathrm{rd}}$ of November, 2017. Reading this 4-slides-per-page handout may be the quickest way to get a tour of this thesis. I hope you will enjoy the tour!

2

Telescope GasP (TGasP)

This chapter gives the rationale for using Telescope GasP (TGasP) implementations for pipeline modules used in ARCwelder [18]. ARCwelder is a design and compilation environment for self-timed dataflow computations. The software for ARCwelder was developed by Willem Mallon at Portland State University between 2010 and 2012. The organization of ARCwelder builds upon results and key lessons learned from the self-timed data-driven compiler effort by Handshake Solutions [43].

The term "telescope," introduced in this thesis, refers to the communication behavior of a pipeline with telescope modules. The forward extension by communication channels with *valid data* and the reverse shortening by communication channels with *no-longer-valid or irrelevant data* are reminiscent of the extension (unfolding) and shortening (folding) of a jointed telescope with sliding tubes — see also Figure 2.1.1. This is particularly obvious for modules with a single data input and a single data output channel:

- **(unfold)** A handshake on the input starts a handshake on the output channel.

- **(fold)** After the output handshake completes, the input handshake completes.

2.1 Why Telescope GasP?

The modules in Willem Mallon's version of ARCwelder are Click modules [18] and most of them use the telescopic handshake relation described earlier, as do most of the original handshake components used by Philips Handshake Solutions [60, 50, 43]. We use Telescope GasP Modules to have a GasP version for such "Telescope Click" modules.

Telescopic handshakes can be used, for instance, to avoid intermediate data storage, which is the key reason why Willem Mallon incorporated them in ARCwelder. They can also be used to bridge long distances [10].

Wires that span a long distance often require amplification. Unidirectional wires, like data wires, can be amplified by inserting repeaters, i.e., buffers or pairs of inverters. Single-track statewires in GasP cannot be amplified using buffers or inverters because they are bi-directional. To amplify these single-track statewires we insert a Telescope GasP module with the following properties:

- The module behaves like a First In First Out (FIFO) buffer module.
- The forward latency in the control flow matches that in the repeated data wires.

The ASYNC 2011 publication by Jo Ebergen et al. [10] presents an design that behaves like a Telescope GasP FIFO module. Insertion of such a module helps reduce or even avoid the relaxation delays for long-range GasP communication

that I presented at ASYNC 2010 [23] and which were also reported separately by Simon Hollis [12].

Figure 2.1.1(top) presents an example of a dataflow FIFO using Telescope GasP modules as repeater modules. Only the Store modules, *Store1* and *Store2*, control latches in the datapath. The Amplify modules, *Amp1* and *Amp2* and *Amp3*, control no data latches. To make this work, the data latch contents feeding the combinational logic must remain stable until the latches fed by the combinational logic have captured the results. To meet this requirement, the Amplify modules keep their predecessors waiting, holding their data valid and stable, until their successors have stored the results of the combinational logic computation, as illustrated in Figure 2.1.1(bottom).

When the leftmost Store module in Figure 2.1.1, *Store1*, initiates a handshake to its successor, *Amp1*, Amplify module *Amp1* initiates a handshake to its successor module, *Amp2*, who then initiates a handshake to its successor, *Amp3*. Meanwhile, the data stored in *Store1* are processed by the combinational logic. After three amplifications, the handshake reaches the rightmost Store module, *Store2*. By the time the handshake reaches *Store2*, the results of the combinational logic computation have arrived at the latch inputs of *Store2*. Module *Store2* now captures these results in its latches, and releases the data-control bundling requirement for its predecessor, *Amp3*, which subsequently releases the bundling requirements for its predecessor, *Amp2*, and so forth, until the bundling requirement on the output channel of the leftmost Store module, *Store1*, is released. At this point, *Store1* may start a new handshake with new data.

Store modules, like *Store1* and *Store2* in Figure 2.1.1(top), are the only modules in ARCwelder that store data. They do so by clocking local latches or flipflops in

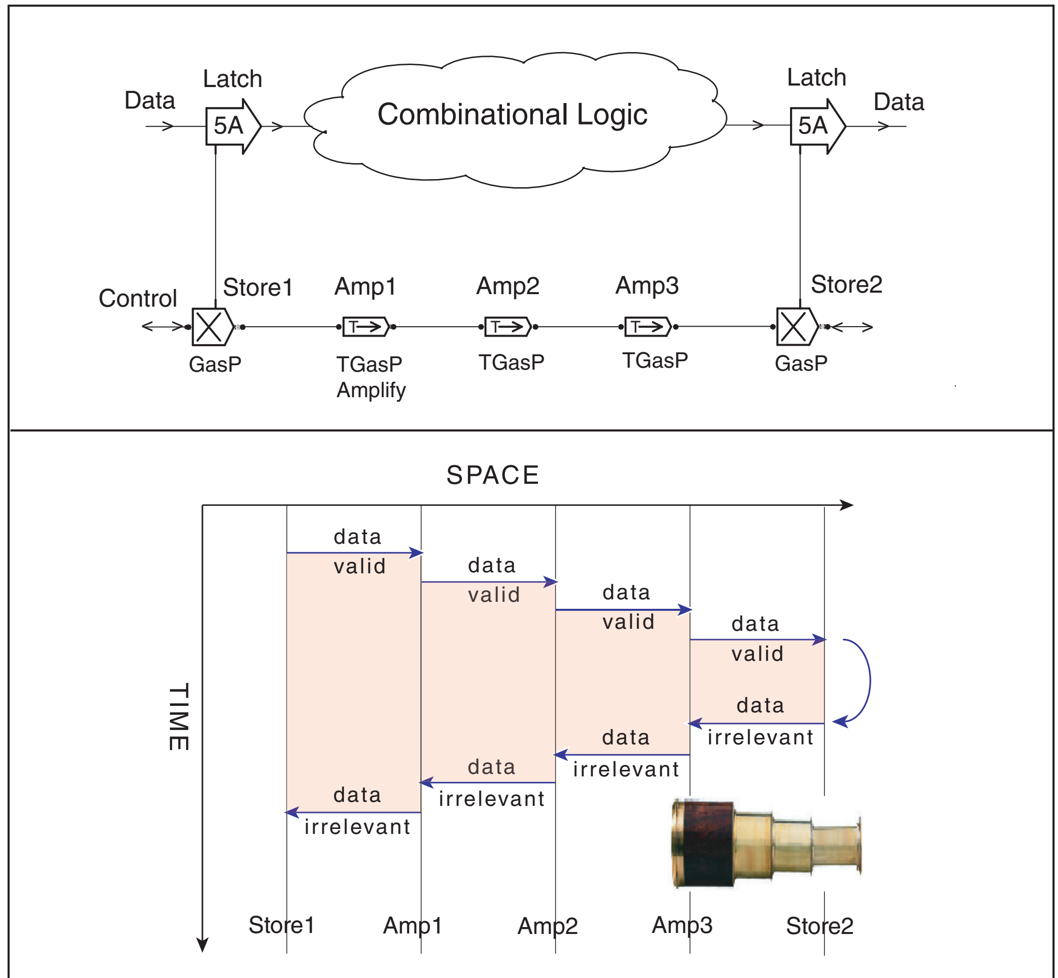


Fig. 2.1.1: (top) Example of a dataflow computation using TGasP Amplify modules. The picture shows the data latches and combinational logic at the top and the control part of the GasP and TGasP modules at the bottom. We assume that the delay through the latches and combinational logic is smaller than the delay through the control. Modules Store1 and Store2 are traditional GasP modules that allow their predecessors to fetch new data on their input channels while their successors process the data on their output channels. Modules Amp1, Amp2 and Amp3 are TGasP modules that keep their predecessors waiting, holding their data valid and stable, until their successors have stored the results of the combinational logic computation. (bottom) Space-Time diagram showing the telescopic handshake behavior of the Amplify modules. Note the forward extension of the handshake channels with *valid data* while the data are processed by the combinational logic between Store1 and Store2. Note the reverse shortening of these same channels with *no-longer-valid or irrelevant data* after the computed results are stored by Store2. This behavior is reminiscent of the extension (unfolding) and shortening (folding) of a jointed telescope with sliding tubes. As reference, a picture of a jointed telescope with sliding tubes has been inserted in the bottom-right corner. Notice its similarity to the shaded area in the Space-Time diagram.

the datapath. As such, Store modules can be used to isolate data computations and datapaths and to interface between different datapath computations. The non-Store modules in Figure 2.1.1 and in the ARC reports in Appendices A, B, and C avoid storing data and clocking latches or flipflops in the datapath.

The ARCwelder compiler lets the designer indicate explicitly when and where data are to be stored. The compiler checks that every loop in the dataflow graph has at least two Store modules; all other modules may be free of storage. ARCwelder also checks that data-control bundling requirements are met on a datapath per datapath basis, from the sending set of Store modules with latches feeding the combinational logic computations in the datapath to the receiving set of Store modules with latches fed by the datapath. It is unnecessary to guarantee data-control bundling requirements on individual, intermediate channel connections. This makes it possible to use time-borrowing techniques to optimize datapaths in their entirety for speed and power [18].

2.2 Traditional versus Telescope GasP

There are two implementation differences between traditional and Telescope GasP. For one, traditional GasP modules have storage, i.e., they control latches in the datapath, where Telescope GasP modules avoid data storage. For instance, signal *cl* in the traditional 6-4 GasP implementation of the Store module in Figure 2.2.1 will clock local latches in the datapath in order to capture and hold data. There is no matching *cl* signal in the Telescope GasP implementation of the Amplify module in Figure 2.2.3.

As a side remark, we can, of course, keep Store signal *cl* in Figure 2.2.1 internal. If we refrain from connecting signal *cl* to any latches in the datapath, the 6-4

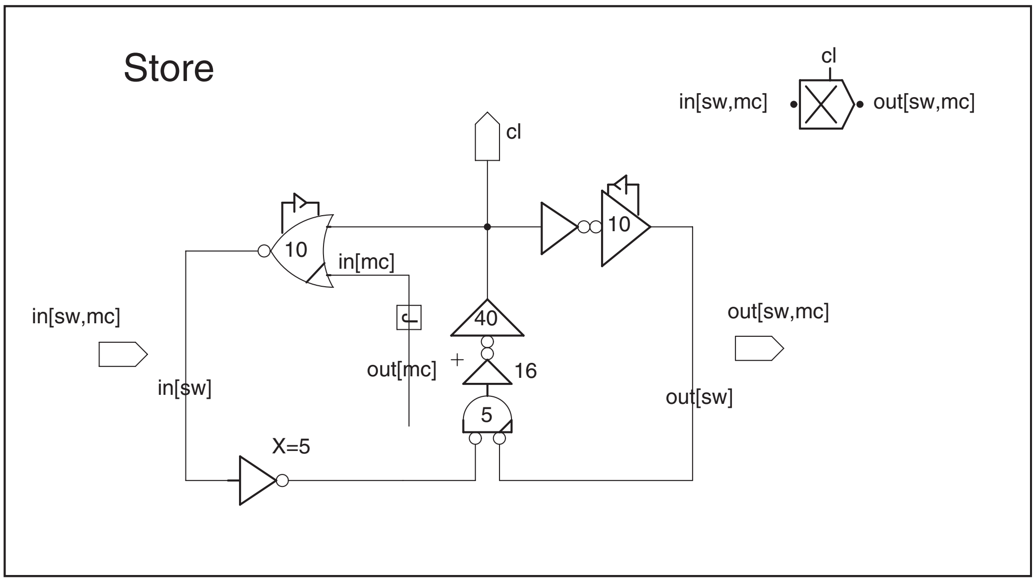


Fig. 2.2.1: Circuit for traditional 6-4 GasP module Store, with input handshake signal and master clear *in[sw,mc]*, output handshake signal and master clear *out[sw,mc]*, and local clock signal *cl* to clock data latches for storing data. We use the top-right icon to represent this GasP Store module.

GasP implementation becomes storage-free. Internalizing signal *cl* is one possible approach for implementing storage-free modules using traditional GasP. So, the fact that the module does or doesn't have storage control is not unique to Telescope GasP. What's unique is the way control-data bundling relations are maintained, which we'll discuss next.

The second and more crucial difference between traditional and Telescope GasP implementation concerns the communication relation between the input and output channels. In traditional GasP, reverse handshakes on the input channels reset while forward handshakes on the output channels start. This leads to maximum concurrency and produces the low cycle times, low latency, and high throughput that are the hallmarks of GasP [55, 23]. This concurrency is clearly visible in the SPICE simulation in Figure 2.2.2.

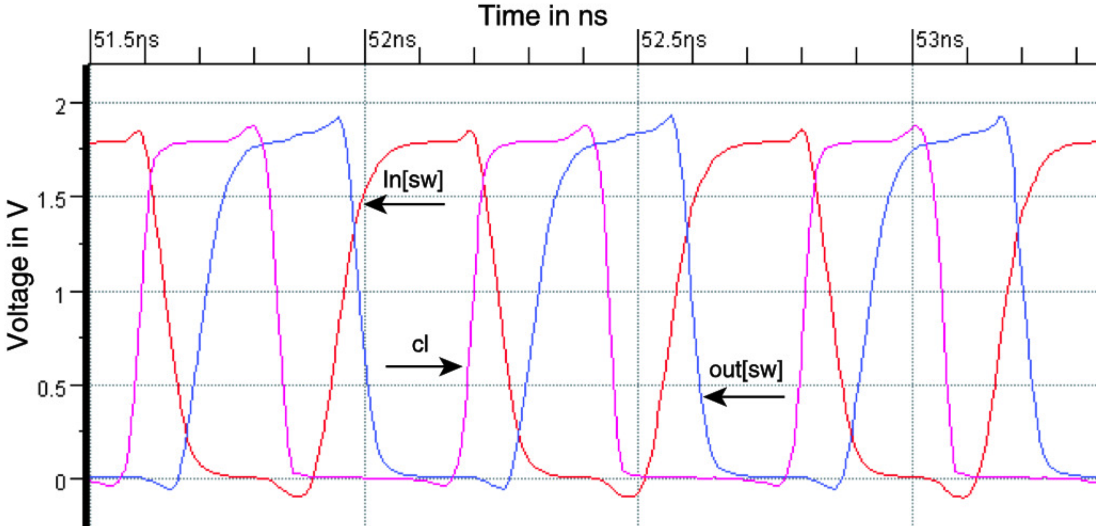


Fig. 2.2.2: SPICE-level simulation of the 6-4 GasP implementation of the Store module in Figure 2.2.1. Red signal *in[sw]* and blue signal *out[sw]* swap their states in parallel and create a high pulse on clock signal *cl*, whenever *in[sw]* is high and *out[sw]* is low. After *in[sw]* has gone low, the predecessor in the simulation test environment switches *in[sw]* back to high. Likewise, after *out[sw]* has gone high, the successor in the simulation test environment switches *out[sw]* back to low. Then, the cycle starts again.

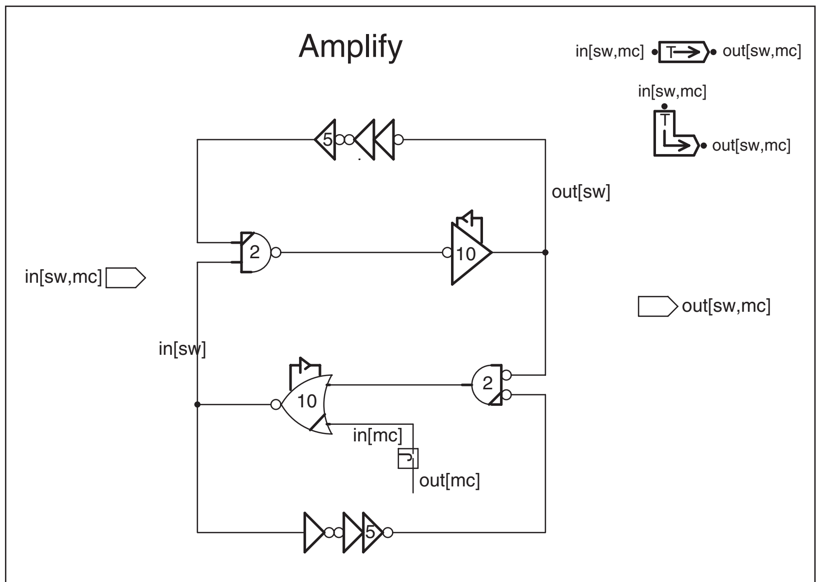


Fig. 2.2.3: Circuit for Telescope GasP module Amplify, with input handshake signal and master clear *in[sw,mc]* and output handshake signal and master clear *out[sw,mc]*. We can use either of the two top-right icons to represent this GasP Amplify module.

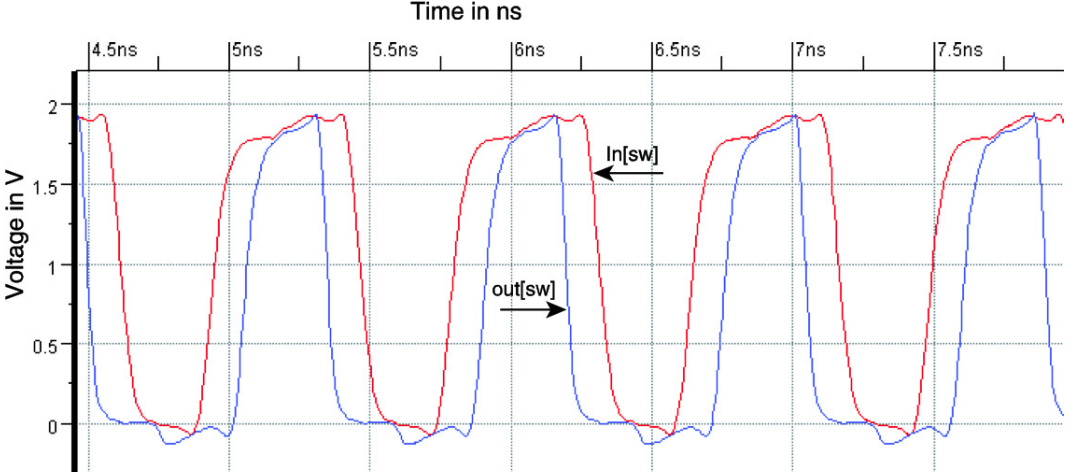


Fig. 2.2.4: SPICE-level simulation of the Telescope GasP Amplify module in Figure 2.2.3. Red signal *in[sw]* and blue signal *out[sw]* have a telescopic handshake relationship: *in[sw]* high causes *out[sw]* to go high, and when *out[sw]* has gone low, only then will *in[sw]* go low. The simulation clearly shows that each high pulse on *out[sw]* is strictly contained within a high pulse on *in[sw]*: the blue signal is always below the red signal.

In Figure 2.2.2, the red input handshake signal *in[sw]* resets, i.e., goes low, at the same time that the blue output handshake signal *out[sw]* starts, i.e., goes high. In Telescope GasP, reverse handshakes on the input channels postpone their reset phase until forward handshakes on the output channels have finished both their start and reset phases. This is clearly visible in the SPICE simulation in Figure 2.2.4: each high pulse on the red input signal *in[sw]* completely encapsulates the high pulse on the blue output signal *out[sw]*.

2.3 Cost of and Alternatives to Telescope GasP

Because each reverse handshake in the dataflow graph postpones its reset phase until the forward handshake is complete, the control-data bundling constraints in Telescope GasP keep their simple form: a high request signal indicates that the data signals have arrived and that the data values are valid and stable. The strictness with which this simple control-data bundling is maintained may be overkill, but it does avoid the need for latching and it avoids the need for keeping intermediate data outside of Store modules. The price for this simplicity, in particular in combination with single-track handshaking, is extra latency on the reverse path. We will explain this below.

Single-track handshake protocols share state. To separate forward from reverse state in a single-track statewire and create different forward and reverse operations during a single-track handshake, one must add at least one gate inversion in each handshake direction. In the case of non-inverting statewires that use the same encoding, e.g., high for *valid data* and low for *no-longer-valid or irrelevant data*, one must add an even number of inversions in each handshake direction. So, for Telescope GasP, it takes at least four inverter delays per hand-

shake cycle to do something different in the forward direction in comparison to the backward, i.e., reverse, direction. One has only to look at the handshake protocol to realize that forward and reverse functionality are always different in Telescope GasP: reverse resets wait until forward resets are done. As a result, each Telescope GasP module adds at least four gate delays to the overall cycle time, just to implement telescoping.[1] A pipeline with 6-4 GasP implementations for the Store module and with Telescope GasP implementations for other modules has a minimum cumulative cycle time of 10+4k gate delays, where k is the maximum number of consecutive Telescope GasP modules between Store modules.

This unavoidable increase in cycle time for inserting single-track style repeater modules may be a disadvantage over design styles that work with multi-track handshake signaling and insert multi-track repeater modules. With separate tracks in each forward and reverse handshake direction, it is possible to add gate delays in one direction but not in the other. See for instance the double-track Click circuit in Figure 3.2.2(top), which has gates in the reverse direction but not in the forward direction. Click uses two-phase handshaking over separate tracks [43]. Consequently, Click datapath designs may not experience as much increased cycle time when they use telescoping as do designs with GasP and TGasP. But even in Click, changes in cycle time will be cumulative for pipelines of telescope modules, because of the telescope unfolding and folding sequence.

Should we conclude that the behavioral implications of telescope handshaking combine adversely with the physical implications of single-track handshaking? If so, then perhaps we should avoid using Telescope GasP and instead continue using traditional 6-4 GasP with its minimum constant cycle time of 10 gate de-

[1] We size transistors using a logical effort model and thus each gate contributes to a unit gate delay.

lays. Using traditional GasP to implement the dataflow design in Figure 2.1.1 may work if we use additional synchronization between the Store modules at the beginning and end of each datapath. The purpose of such begin-end synchronization is to keep the data latches that feed the combinational logic valid and stable until the latches fed by the combinational logic have captured the computed results. This could provide an alternative solution for using a telescope communication protocol when the goal is to avoid intermediate data storage. It would be a dubious solution when the goal is to bridge long distances. Also, this alternative solution would change the dataflow graph input to ARCwelder, because the dataflow graph would contain extra channel connections and it would use a different handshake signaling sequence. As such, this alternative solution approach is of little help when it comes to mapping *existing* dataflow designs in ARCwelder.

I pursue Telescope GasP for a simple reason: to have a "drop-in," i.e., module-by-module, GasP-style replacement for the dataflow designs already present in ARCwelder and implemented currently using Click Store modules and "Telescope Click" non-Store modules. Assuming that we can make the forward latency in Telescope GasP match the delay needs for control-data bundling, e.g., assuming that the forward latency in the control path matches the delay of the combinational logic, then the art of reducing the datapath cycle time is really in reducing the reverse latency. We can reduce the reverse latency in a datapath by using a fast reset strategy on the reverse handshakes. Section 2.4 contains an example of how this can be done for the Amplify module. The design solutions presented in Appendices A, B, and C show how one can do this for other existing ARCwelder modules that use telescope handshakes.

2.4 Designing Telescope GasP Modules

Our actual designs of the Telescope GasP modules can be found in Appendices A, B, and C. Here, in this section, we give only a flavor of the design approach. We explored three different approaches in designing Telescope GasP modules. The first approach uses separate self-resetting loops for input and output channels. The Amplify module in Figure 2.2.3 is an example of this first design approach. It has two self-resetting loops: the loop at the top is used to start and stop the high drive for *out[sw]*, while the loop at the bottom is used to start and stop the low drive for *in[sw]*. The second approach shown in Figure 2.4.1 shares the self-resetting loops. It has one self-resetting loop, with separate taps to start and stop the high respectively low drives for *out[sw]* and *in[sw]*. The second design approach is based on the repeater design by Jo Ebergen et al. [10] but we generalized the approach to other modules beyond Amplify. The third design approach shown in Figure 2.4.2 uses separate self-resetting loops for handshake signals as used in the first design, but it cross-couples the loops to improve timing margins. This is explained in more detail further on.

All three approaches have their advantages and disadvantages. The advantage of the first approach used in Figure 2.2.3 is that its control-data bundling constraints are less strict than those for the second approach used in Figure 2.4.1. This is not immediately obvious from the Amplify design, but it will become obvious when we get to the data-controlled Distribute modules. For details, see Appendix B.

The advantage of the second approach used in Figure 2.4.1 is that it has timing margins that we feel are safe — safer than those in the first approach. The caption of Figure 2.4.3 contains a detailed explanation of the Amplify operation

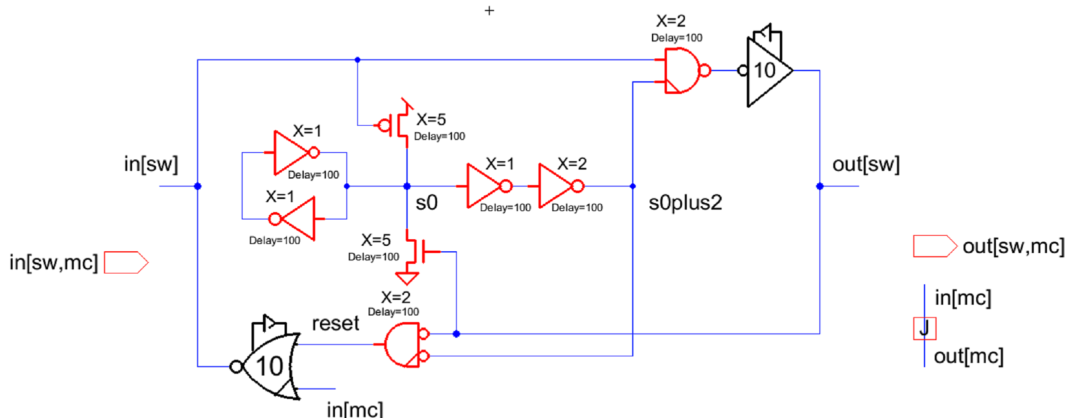


Fig. 2.4.1: Our second alternative Telescope GasP circuit implementation for module Amplify. It differs from the first one in Figure 2.2.3, because it shares the self-resetting loops between the input and output handshakes insofar possible.

and safe timing margins in the second design approach.

In contrast to the safe timing margins in the second design approach, the first design approach in Figure 2.2.3 has two marginal relative timing constraints:

1. After *in[sw]* rises, it takes 2 gate delays before *out[sw]* rises and disables the inverted input AND gate in the lower self-resetting loop, while it takes 3 gate delays for the high *in[sw]* signal to enable that same AND gate via the lower self-resetting loop. To guarantee a telescope relation between *in[sw] and out[sw]*, the inverted input AND gate must sense the disabling *out[sw]* transition before it senses the enabling *in[sw]* transition. The timing margin between the two is only 1 gate delay.

2. After *out[sw]* falls, it takes 2 gate delays before *in[sw]* falls and disables the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the low *out[sw]* signal to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *in[sw]* and *out[sw]*, the NAND gate must sense the disabling *in[sw]* before

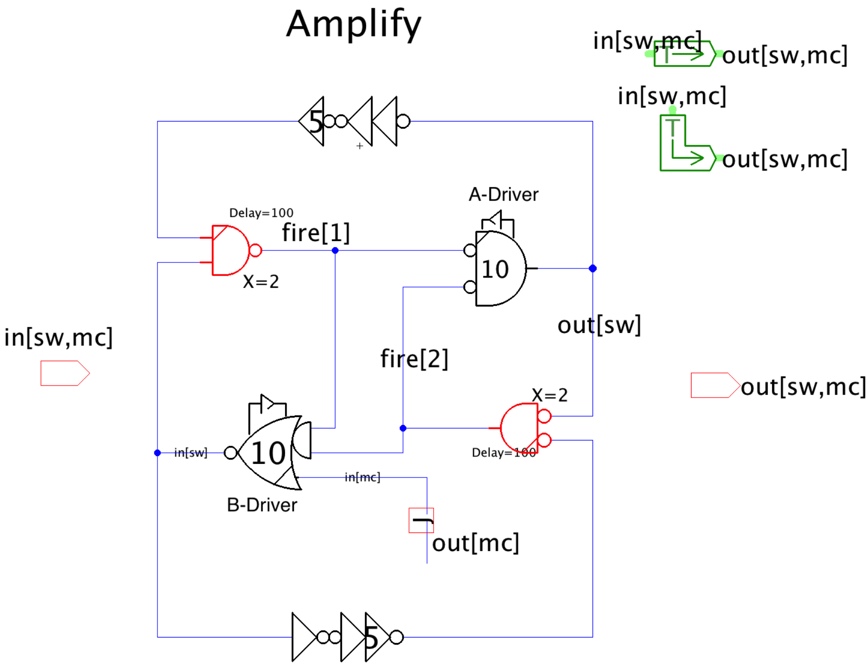


Fig. 2.4.2: Our third alternative Telescope GasP circuit implementation for module Amplify, similar to the first one in Figure 2.2.3 but with better timing margins.

it senses the enabling *out[sw]* transition. The timing margin between the two is only 1 gate delay.

A timing margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can make both margins as safe as the timing margins deployed in the second approach by adding extra circuitry to delay the transition that must arrive later by two additional gate delays. Other transitions remain unaffected. For instance, to delay a transition from *sin* low to *sout* high we insert the left-hand circuit of Figure 2.4.4, and to delay a transition from *sin* high to *sout* low we insert the right-hand circuit of Figure 2.4.4.

The two delay circuits in Figure 2.4.4 add two gate delays to the targeted *sin*-to-*sout* transition. Depending on the actual cycle time of the module in the given datapath, we can extend this further to four additional gate delays or more, with-

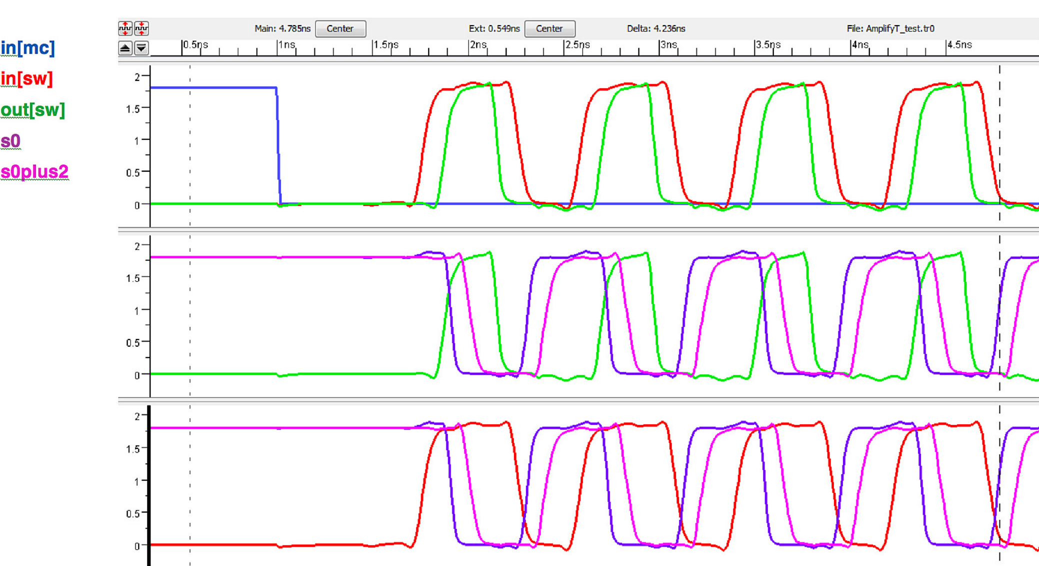


Fig. 2.4.3: SPICE-level simulation of module Amplify in Figure 2.4.1.
• The top window shows the telescopic handshake relation between *in[sw]* and *out[sw]*.
• The middle window shows the simulated timing margins from *out[sw]* high to s0 low (1 gate delay) and to s0plus2 low (3 gate delays) and to stopping the high drive on *out[sw]* (5 gate delays). The *in[sw]* reset timing margin of 3 gate delays that we see when going from *out[sw]* high, disabling the inverted input AND gate, to s0plus2 low, enabling the AND gate, is large enough to guarantee the telescopic relation between *in[sw]* and *out[sw]*. It is also small enough to fit within the high pulse width of 5 gate delays for *out[sw]* in order to maintain throughput. The remaining 2 gate delays after s0plus2 goes low are used to stop the high drive on *out[sw]* so the receiver can reset *out[sw]* without a fight conflict.
• Similar to the middle window, the bottom window shows the simulated delay margins from *in[sw]* going low to s0 high (1 gate delay) and to s0plus2 high (3 gate delays) and to stopping the low drive on *in[sw]* (5 gate delays). The *out[sw]* set timing margin of 3 gate delays that we see when going from *in[sw]* low, disabling the NAND gate, to s0plus2 high, enabling the NAND gate, is large enough to guarantee a telescopic handshake relation between *in[sw]* and *out[sw]*. It is also small enough to fit within the low pulse width of 5 gate delays for *in[sw]* in order to maintain throughput. The remaining 2 gate delays after s0plus2 goes high are used to stop the low drive on *in[sw]*, so the sender can set *in[sw]* without a fight conflict.

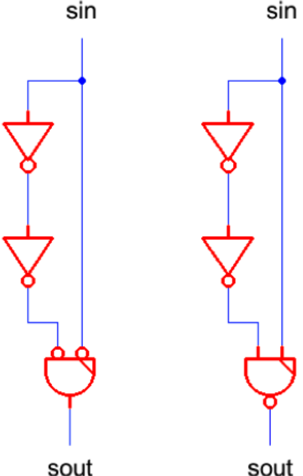


Fig. 2.4.4: Delay circuits to increase the timing margin from *sin* low to *sout* high by two gate delays (left), and to increase the delay from *sin* high to *sout* low by two gate delays (right). Other path delays remain unchanged.

out losing throughput. These two types of delay circuits are sufficiently generic for our needs, and have the following properties:

- The target transition is slowed down by sufficient margin.
- The target transition is fast enough to sustain maximum throughput.
- The delays of the other transitions remain unchanged.

Adding delay circuits to improve the timing margin has one limitation. The delay circuits are asymmetric in nature, which limits the maximum gate delays one can obtain by inserting such delay modules. In this thesis, the maximum number of gate delays is 5, as limited by the 5-gate delay pulse width of 6-4 GasP.

An alternative to adding delay circuits in the first design is to use the third design approach shown in Figure 2.4.2, which has the following relative timing constraints:

1. After *in[sw]* rises, it takes 1 gate delay to disable the B-Driver gate in the lower self-resetting loop, while it takes 4 gate delays for the high *in[sw]* signal to enable this B-Driver gate via the lower self-resetting loop. To guarantee a telescope relation between *in[sw]* and *out[sw]*, the B-Driver gate must sense the *in[sw]*-based disabling transition before it senses the *in[sw]*-based enabling transition. The timing margin between the two is 3 gate delays, which we consider safe.

2. After *out[sw]* falls, it takes 1 gate delay to disable the A-Driver in the upper self-resetting loop, while it takes 4 gate delays for the low *out[sw]* signal to enable the same A-Driver via the upper self-resetting loop. To guarantee a correct handshake relation between *in[sw] and out[sw]*, the A-Driver gate must sense the *out[sw]*-based disabling transition before it senses the *out[sw]*-based enabling transition. The timing margin between the two is 3 gate delays, which we consider safe.

The key advantage of the third design approach in Figure 2.4.2 is that both relative timing constraints have a safe timing margin of 3 gate delays. A disadvantage of the third design approach is that the complexity of the A-Driver and B-Driver gates increases as each gate gets an additional input.

2.5 Latency Reduction in the Reverse Direction

In addition to the single module test configuration used for Figure 2.4.3, we also tested the behavior of the Amplify module in a datapath configuration consisting of a First In First Out (FIFO) configuration of Amplify modules. The purpose of this second test configuration was to show how one could generate a fast

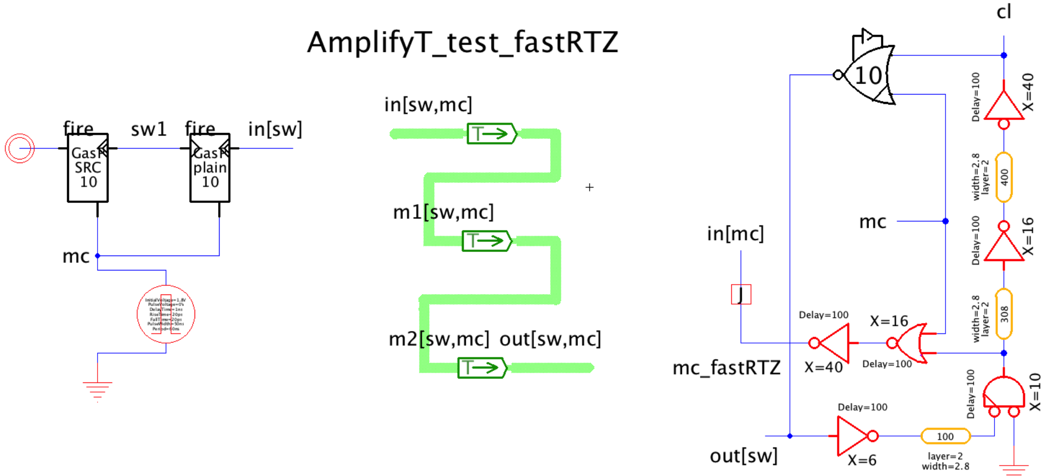


Fig. 2.5.1: Fast reset configuration to reset concurrently the three Amplify modules in the middle First In First Out (FIFO) design.

reset signal to shorten the reverse latency in the datapath. This second test configuration follows in Figure 2.5.1.

Figure 2.5.1 shows a sender FIFO configuration on the left, designed using traditional 6-4 GasP modules. The sender FIFO feeds a datapath consisting of another FIFO configuration with three Amplify modules, designed in Telescope GasP using the circuit implementation of Figure 2.4.1. The right-hand side shows the receiving half of a Store module, designed in 6-4 GasP using the implementation of Figure 2.2.1. We deleted the sending half from this Store module because we don't need it, and we added extra circuitry to generate a fast reset signal, called *mc_fastRTZ*.

When the input handshake signal to the Store module on the right rises, i.e., *out[sw]* goes high, the module generates a 5-gate-delay high pulse on *mc_fastRTZ*, which is distributed to reset concurrently all outstanding handshakes for the Amplify modules in the middle FIFO. This works, as can be seen

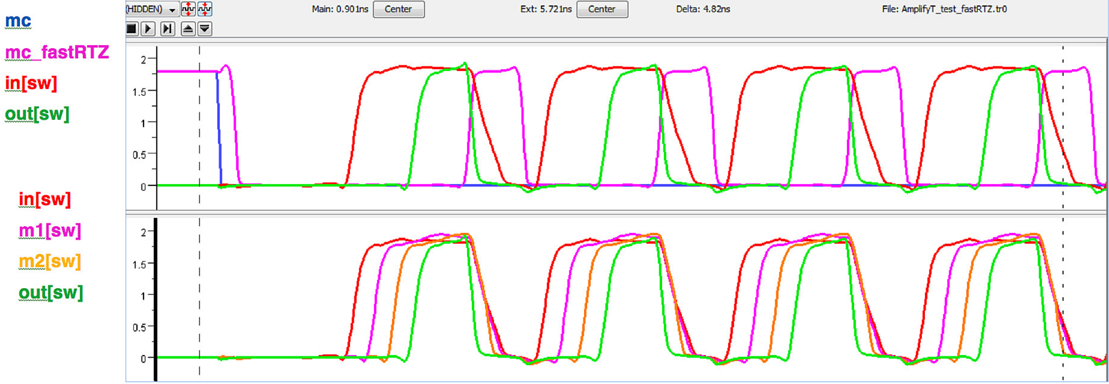


Fig. 2.5.2: Fast reset SPICE-level simulation results for the test configuration in Figure 2.5.1.
• The top window shows how the pink high pulse on the fast reset signal, *mc_fastRTZ*, resets concurrently both the green handshake on *out[sw]* of the third Amplify module and the red handshake on *in[sw]*. The reset slope for *out[sw]* is steeper than the reset slope for *in[sw]* because: (1) *out[sw]* is reset first by *mc_fastRTZ* and two gate delays later also by *out[sw]* via the negated input AND gate in the third Amplify module and the strong cl-to-out[sw] driver in the Store module, while (2) *in[sw]* is reset first by *mc_fastRTZ*, and two gate delays later by a weaker driven *m1[sw]*.
• The bottom window shows the beginning and end of each handshake over the initial, intermediate and final statewires *in[sw]*, *m1[sw]*, *m2[sw]*, and *out[sw]* in the Amplify queue. The handshakes at the output channel of each module start 2 gate delays after the handshakes at the input channel. But thanks to the fast reset signal, *mc_fastRTZ*, shown in the top window, all outstanding handshakes end at the same time. The steepness of the reset slopes for *in[sw]*, *m1[sw], m2[sw]*, *out[sw]* depends on how many modules there are between the statewire and the Store module at the end of the datapath: the fewer modules in-between, the steeper the slope. This is because the second drive wave for resetting each handshake comes from the successor module, which is strongest when coming from the Store module. In the above text for the top window, we already explained the slope differences for the falling transitions on *in[sw]* versus *out[sw]*. The bottom simulation window clearly shows that differences in slope peter out quickly: the slope differences for the falling transitions on *in[sw]*, *m1[sw]*, and *m2[sw]* are significantly less prominent.

from the SPICE simulation in Figure 2.5.2. The cycle time for the Amplify FIFO without a fast reset signal would be 10 + 3*4 = 22 gate delays. With the fast reset signal we obtain a cycle time of 10 + 3*2 = 16 gate delays. These results are same with the Amplify implementations of Figures 2.2.3 and 2.4.2.

2.6 Summary and Conclusions

This chapter introduced Telescope GasP, or TGasP, to have a module-by-module GasP-style replacement for the dataflow designs already present in ARCwelder and implemented currently using Click Store modules and “Telescope Click” non-Store modules. The idea is that, instead of mapping a Store module to a Click Store circuit, ARCwelder can now map a Store module to a GasP Store circuit. And, instead of mapping a non-Store module to a specific “Telescope Click” circuit implementation, ARCwelder can now map a non-Store module to a TGasP circuit implementation. In the next chapter, I will discuss the ARCwelder compiler extensions for GasP and TGasP.

I introduced the term “telescope” for the communication behavior of a “telescope” module, because the forward extension of its communication channels with *valid data* and the reverse shortening with *no-longer-valid or irrelevant data* are reminiscent of the extension and shortening of a jointed telescope with sliding tubes, as illustrated in Figure 2.1.1.

Telescope handshake modules can be used to avoid intermediate data storage, which is the key reason why Willem Mallon incorporated them in ARCwelder. They can also be used as repeater modules to bridge long distances.

An alternative solution to avoid intermediate storage — over short distances —

would be to maintain a parallel communication protocol, such as used by Store modules, and avoid storing intermediate data but to keep the data stable and valid for as long as needed by adding extra — short-distance — synchronization channels between the Store modules at the beginning and end of each datapath.

Telescope GasP suffers from latency in the reverse direction of the dataflow. Section 2.5 and Appendices A, B, and C present a fast reset solution to ameliorate this problem. There are similarities between implementing these fast reset solutions for Telescope GasP and implementing a datapath begin-to-end synchronization scheme for traditional GasP. Further investigations of these similarities are outside the scope of this thesis. Also outside the scope of this thesis are our continued investigations as to where and by how much we might want to change the forward and reverse latencies in Telescope GasP. The low 2-gate-delay latencies used in this thesis result in a myriad of custom driver designs.

2.7 My Contributions for Chapter 2

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

- I extended the GasP-style circuit family with Telescope GasP (TGasP) modules. I developed a representative set of TGasP modules for control-driven and data-driven dataflow control operations: fork, join, branch, case selection, non-arbitrated and arbitrated merge, and repetition. The corresponding TGasP module designs follow in the ARC reports in Appendices A, B, and C. All modules have been sized correctly and have been simulated with SPICE to validate their behaviors.

- I developed three different circuit design approaches to implement each Telescope GasP module, and explored their advantages and disadvantages in terms of design complexity and timing.

3

Silicon Compilation for Click and GasP

To design any computer system of substance, we need design tools [7][14], preferably a silicon compiler – with a user interface to enter the design at a high enough level so we can focus on algorithmic correctness before we map it to a circuit implementation, and with a suite of analysis tools so we can evaluate the various implementation options in terms of energy, throughput, latency, and area. Few silicon compilers are available that support self-timed design. Some of the best known ones are proprietary, owned by companies like Tiempo [5] and Fulcrum [3].

This chapter summarizes the changes made to the ARC's silicon compiler, ARCwelder, so it compiles dataflow designs to both Click and GasP-style implementations.

ARCwelder was developed by Willem Mallon, between 2010 an 2012, when Willem worked at the Asynchronous Research Center (ARC) at Portland State University. The compiler maps dataflow designs to VLSI circuits and – ultimately – to silicon. The name of the compiler reflects how, at the user interface, ARCwelder welds together both text and graphics to convey the intent of the design.

The organization of ARCwelder is based on that of the last compiler developed by Philips Handshake Solutions [43]. The compiler focuses on dataflow designs, as opposed to the control-oriented focus of previous compiler generations developed and supported by Handshake Solutions and the prior Philips Tangram team [42, 60]. The reason for this focus shift was caused by a shift in applications requested by the customers of Handshake Solutions. Originally, most applications were in the area of low power, automotive, smart cards, and distributed control. But more and more customers were seeking higher throughput applications in the signal-processing domain. After delivering the asynchronous ARM996HS in 1996, which was licensed through ARM Ltd, the design engineers at Handshake Solutions realized that this type of design stretched the compiler capabilities to the limit in terms of circuit latency and throughput. Meanwhile, the test engineers had come to the conclusion that, despite intense collaboration between Handshake Solutions and the Computer Aided Design (CAD) tool companies they worked with, the CAD tools were largely incapable of dealing with latch-based designs. As a result, most of the latches had to be replaced by flipflops to permit static timing analysis, fault simulation, and test generation. The engineers developed a new circuit family, called Click, which uses flipflops instead of latches. All loops in Click go through flipflops. The data are tightly coupled to the control flow to accommodate pipelined applications with high throughput dataflows while saving power and energy. Click is the most synchronous asynchronous self-timed circuit family we know of. Commercial CAD tools for static timing analysis, fault simulation, and test generation have an easier task dealing with Click circuits than with most other self-timed circuits. For Handshake Solutions, supporting Click required the development of a new compiler [43]. Neither Click nor its compiler were patented, because by that time Philips was closing down Handshake Solutions.

At the ARC, we're trying to save some of the key insights and knowledge built up by Handshake Solutions, starting with Click and its compiler. Willem Mallon has played a key role in kick-starting this effort, and is still dedicated in continuing it through his Dutch consulting company, Third Party B.V., for hardware and software compilers. ARCwelder is available through a Berkeley Software Distribution (BSD) permissive free software license. A major hurdle with ARCwelder, after Willem left, was that there were no instruction manuals for either the Graphical User Interface (GUI) or the compiler source code. Ivan Sutherland has taken over the GUI implementation, so the GUI is now covered. Together with Hoon Park, Anping He, Navaneeth Jamadagni, Ivan Sutherland, and Marly Roncken, I looked at the compilation and static and dynamic timing analysis parts. We wrote cheatsheets for how to use the GUI, how to compile a GUI design to a Click circuit, how to model and simulate the circuit in Verilog, and how to run static timing analysis and fix timing constraints [39, 40, 35].

3.1 Existing ARCwelder Compiler

Figure 3.1.1 gives an overview of the various capabilities of ARCwelder, showing the functions supported by the compiler and their position in the design flow. I use blue-colored boxes to indicate the functions, and pink-colored ones to indicate the input files used by a function and the output files produced as a result of executing the function.

During their time at the ARC, Navaneeth Jamadagni and Willem Mallon developed ARCwelder interfaces to commercial CAD tools for placement and routing (Place&Route) and back-annotation of the Place&Route results to get more accurate simulation results in terms of power, latency, and throughput. As part of

his PhD research [15, 16], Navaneeth Jamadagni investigated how best to use commercial CAD tools to optimize and implement arithmetic datapath computations.

In the following subsections, I will give an overview of the key stages in the ARCwelder compiler flow: Graphical User Interface, Parser, Static and Dynamic Validation.

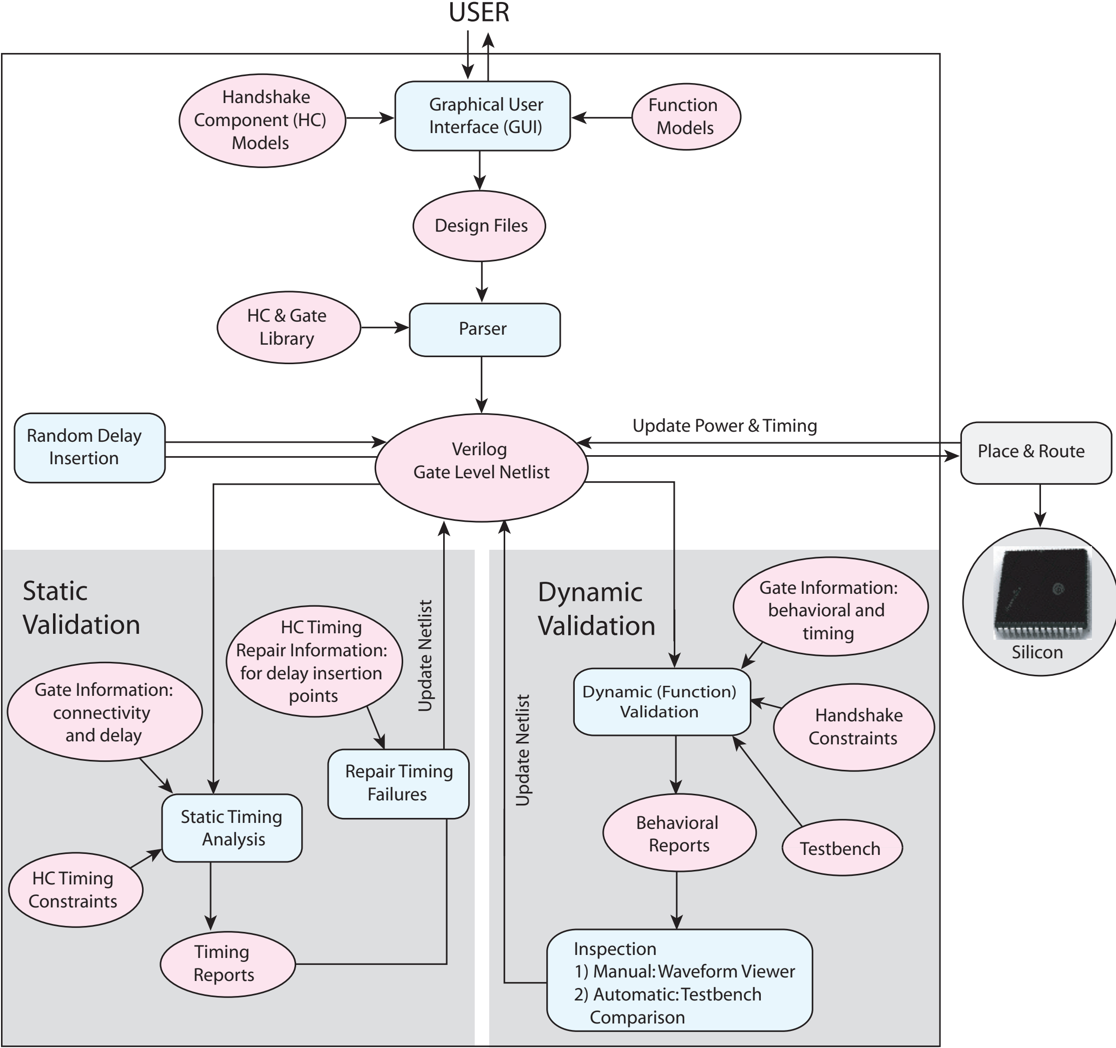


Fig. 3.1.1: ARCwelder organization, with the key compiler functions colored blue and the input and output files used and produced by each compiler function colored pink.

3.1.1 Graphical User Interface

Each ARCwelder design starts at the Graphical User Interface (GUI). The designer uses the GUI to create designs. Each design is a network of handshake modules connected by handshake channels. The designer creates the network topology of modules and channels — *graphically*. The designer provides the data operations, data types, and handshake behaviors for the modules and channels — *textually*. The designer also uses the GUI to inspect the correctness of the design by simulating the handshake behavior.

3.1.2 Parser

The second compiler stage in ARCwelder is the Parser. The Parser maps a GUI design to a circuit. Users can choose the type of circuit family that they wish to map to.

3.1.3 Static Validation

The third compiler stage in the ARCwelder design flow is Static Validation. The purpose of Static Validation is to check whether the design meets the timing constraints for correct communication and data transfer. The generation of these timing constraints is the subject of Hoon Park's PhD research [39, 41].

3.1.4 Dynamic Validation

The fourth compiler function in the design flow is Dynamic Validation. The purpose of dynamic validation is to check the functional correctness of the design

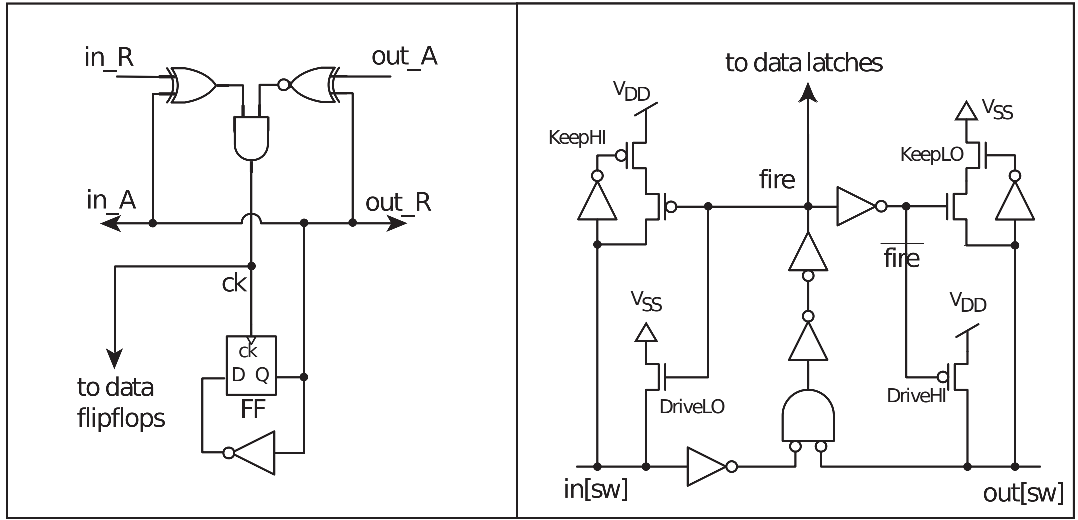


Fig. 3.2.1: Implementation of a Store module in Click (left) and GasP (right).

through simulation.

3.2 ARCwelder Compiled Circuits: Existing versus New

At the ARC, we use both Click and GasP circuits. GasP originated from Sun Microsystems Laboratories [55] in the United States. Click originated from Philips Handshake Solutions [43] in The Netherlands. The current version of ARCwelder compiles dataflow designs to Click circuits. To make ARCwelder usable for both types of circuits that we work with at the ARC, I extended the compiler so it supports not only Click [43] but also GasP [53]. Click and GasP circuits have a different architecture and a different handshake behavior. This section explains the key differences that mattered most in adapting the compiler to fit both families.

Figure 3.2.1 shows two implementations of a Store module, in Click respectively GasP. Before going over their behavior, let's first look at the architecture of the designs.

- The Click version uses two wires per handshake channel, a request wire to start the handshake, and an acknowledge wire to signal its completion. For instance, handshake channel *in* has request wire *in_R* and acknowledge wire *in_A*.

  The GasP version uses a single bi-directional wire per handshake channel, which acts as request when high, and as acknowledge when low. For instance, handshake channel *in* has a single bi-directional wire *in[sw]*.

- In Click, the state is stored in data flipflops and in the control flipflop, *FF*.

  In GasP, the state is stored in data latches and on the statewires, *in[sw]* and *out[sw]*, and their half-drivers and half-keepers. In Figure 3.2.1(right), the half-keeper for keeping the statewire high, *KeepHI*, is drawn at the module side that drives the statewire low, *DriveLO*. Likewise, the half-keeper for keeping the statewire low, *KeepLO*, is drawn at the module side that drives the statewire high, *DriveHI*. This way, the half-keeper that keeps the statewire one way can be shut off immediately when the statewire is driven the other way, thereby avoiding a drive fight. Each GasP statewire in Figure 3.2.1 is driven for 5 gate delays. During that time, the drive is handed over to the half-keeper at the other end of the channel. After 5 gate delays, the half-driver at the other end of the channel may drive the statewire the other way.

- Both Click and GasP have self-resetting loops, and both generate a high pulse local clock per handshake, called *ck* in Click and *fire* in GasP — see Figure 3.2.1. The high pulse temporarily renders the storage elements in the datapath — flipflops in Click, latches in GasP — transparent, and thus copies the data from the input channel to the output channel.

As far as their behavior goes: **each Click and GasP Store module acts when**

- **its input channel has valid data** —
  i.e., when *in_R* differs from *in_A* in Click, and when *in[sw]* is high in GasP, **and**
- **its output channel has space** —
  i.e., when *out_R* matches *out_A* in Click, and *out[sw]* is low in GasP.

When it acts, it raises its local clock — *ck* for Click in Figure 3.2.1 and *fire* for GasP — **resulting in the following three concurrent actions:**

1. **The data latches or flipflops capture the data**, passing data from *in* to *out*.
2. **Channel *in* is declared empty** – its handshake completes:
   - In Click, *in_A* flips and its value now matches with the value on *in_R*.
   - In GasP, *in[sw]* goes low.
3. **Channel *out* is declared full** – its handshake starts:
   - In Click, *out_R* flips and its value now differs from *out_A*.
   - In GasP, *out[sw]* goes high.

The changes on the channel wires activate two internal reset loops that reset the local clock again to low. At this point, the states of the control wires in each channel are kept by flipflop *FF* in the Click module and by the half-keepers in the GasP module.

All our current GasP modules control data latches in the datapath and overlap the handshake completion on channel *in* with the handshake start on channel *out*, while handing over the data from *in* to *out*.

This is not the case for all Click modules used in ARCwelder. In fact, the Store module is the only Click module with data storage and overlapping completion and start of input-output handshakes. All other Click modules in ARCwelder

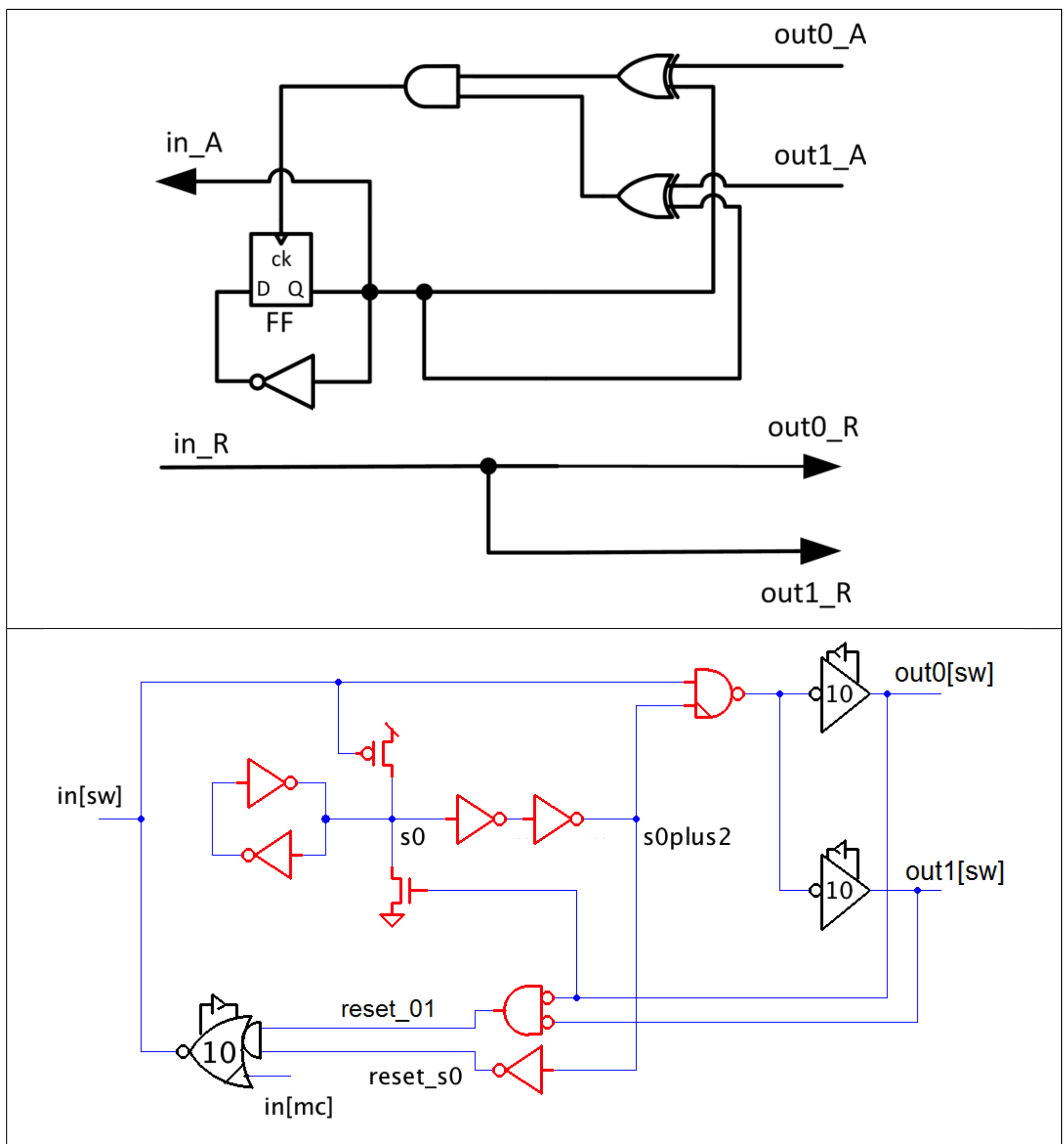


Fig. 3.2.2: Fork module implemented in Click (top) and in TGasP (bottom).

transfer but refrain from storing data. While transferring data, a non-Store module keeps its input channels suspended in their handshake communications until the output channels have completed theirs.

Figure 3.2.2 (top) gives an example of such a non-Store Click module: a 1-input

2-output Fork module. Click circuits are initialized with all handshake signals set to low. From there, the handshake behavior for the Fork module proceeds as follows. A change on input request signal *in_R* leads to a similar change on the out0_R and out1_R request signals for the two output channels of the module. The module then waits until both output successor modules have completed their computations and terminated the output handshakes by flipping *out0_A* and *out1_A*. The changes on *out0_A* and *out1_A* make the flipflop's clock pin (*ck*) high, which flips the value at the flipflop's output pin (Q), thus causing *in_A* to flip and terminate the handshake on input channel, *in*. The termination of the input handshake also causes the flipflop's clock pin to reset low via the two self-resetting internal loops through the XOR and AND gates, in preparation for the next handshake cycle. Note that the Fork's handshake behavior acts like a telescope:

- it unfolds its channel handshakes from left to right (*in* to *out*)
  — declaring them full with valid data,
- and it folds them back from right to left (*out* to *in*)
  — declaring them empty with no-longer-relevant data.

In Chapter 2, we introduced GasP-style modules that operate using such telescopic handshakes. We called this GasP-style extension "Telescope GasP" — "TGasP" in short.

A TGasP version of the Fork module follows in Figure 3.2.2 (bottom) – see also Figure 15 in Appendix A. It uses an internal set–reset latch to remember whether the next action is in the forward or in the reverse direction. Just like in traditional 6-4 GasP, it drives the statewires for 5 gate delays before handing the drive back to the statewire keepers. Like Click, this TGasP implementation assumes that all the statewires start low (empty).

A representative set of TGasP modules, with correctly sized gates and transistors and with SPICE simulations, can be found in Appendices A, B and C. I have used these TGasP modules and the GasP Store module in Figure 3.2.1 to investigate how to support GasP-style circuits in ARCwelder. Given that ARCwelder already works for Click, it made sense to start with the same module base. Section 3.3 discusses the ARCwelder adaptations that I made to compile dataflow designs to both Click and (T)GasP circuits.

3.3 ARCwelder for Click and (T)GasP

To make ARCwelder usable for the type of circuits that we work with at the ARC, we extended the compiler so it supports not only Click [43] but also GasP-style circuits [53]. Our goal is to use the same GUI design as was used for the Click implementation, and just re-map the modules and channels in the design to (T)GasP instead of Click.

The subsections below discuss changes in each of the four compiler stages shown in Figure 3.1.1. These changes allowed me to create a first – not necessarily elegant but operational – version of ARCwelder that supports both Click and (T)GasP.

3.3.1 Graphical User Interface Changes

Each ARCwelder design starts in the GUI. The designer uses the GUI to indicate which handshake control modules and data operations to use, and how to connect them. He or she can use a standard set of modules, or define new ones in the GUI. The designer displays the modules and their connections graphically,

Fibonacci Design

```
typedef bool[16] FibWord;

function plus
( x : FibWord, y : FibWord): FibWord{ x + y }

node lastFib: fullstore(
.in[0](FibWord),
.out[0](FibWord),
.out[1](FibWord),
.initial(16'b0000000000000001))

node next_to_lastFib: fullstore(
.in[0](FibWord),
.out[0](FibWord),
.initial(16'b0000000000000000))

node sum: join(
.in[0](FibWord),
.in[1](FibWord),
.out(FibWord),
.func(plus))

edge con0, con1, con2, con3, con4, con5;
```

Fig. 3.3.1: GUI design for a Fibonacci number generator, generating the sum of the last two numbers in a sequence, primed here with 0 and 1, and generating 1, 2, 3, 5, etcetera. The design contains two fullStore modules, *lastFib* and *next_to_lastFib*, one Store module, *nextFib*, and one Join, *sum*. Each fullStore module starts with data, Store starts empty. The text files fill in the details on data types (16-bit words, representing a number), initial data settings (0 for *next_to_lastFib*, 1 for *lastFib*), and the data operation in the Join ($+$).

but provides the data operations, data types, and handshake behaviors of the modules in text format.

Figure 3.3.1 shows an example of a GUI design for a Fibonacci number generator. The design contains one Store module, two fullStore modules and a Join module, connected by handshake channels. Text snippets with data operations, data types and initial data settings are shown on the right-hand side of Figure 3.3.1. The designer can use the GUI to inspect the correctness of the design by simulating the handshake behavior, event by event, and by providing data inputs and validating data outputs handshake by handshake.

3.3.2 Parser Changes

The Parser maps the GUI design files to circuit design files that can be simulated in Verilog and analyzed and mapped to silicon. The Parser maintains the design hierarchy of the GUI-level design. Originally, the Parser replaced each handshake module by a corresponding Click circuit implementation in Verilog, each handshake channel by a pair of request-acknowledge wires, and each data operation by a functional Verilog model.

In the extension, the Parser supports mappings to (T)GasP circuits. It maps each handshake module to a corresponding GasP or TGasP circuit implementation in Verilog, each channel to a single bidirectional request-acknowledge wire, called a "statewire," and each data operation by a functional Verilog model. This sounds quite similar to mapping a GUI design to a Click implementation, and it is — except that:

- The bidirectional wire mappings in (T)GasP required additional code and the use of *inout* Verilog signals.
- In Click, all channels are initialized low. In (T)GasP, we view channels as the key state-holding elements of the module, and we may initialize them high or low. This is not a big deal at the Verilog netlist level, but it has additional implications for the dynamic validation code used later in the design flow. During dynamic validation, we check that the data sent over the channel is stable during each handshake, i.e., from the handshake's request to its acknowledge.
- The Parser uses generalized Click modules. The generalized modules support arbitrary numbers of handshake channels for data input and data

output. As a result the Verilog implementations for the various Click modules may vary significantly in area, latency and throughput.

This contrasts with (T)GasP modules, which we design with great care to obtain the same forward latency, the same reverse latency, and the same throughput.

As a result, (T)GasP modules can be generalized to some extent with more input and output channels, but not as extensively as Click modules. This has not been an issue yet because the existing designs that went through ARCwelder have a limited number of input-output channels per module. In the future we may want to change a design at the GUI level, depending on whether we map it to Click or (T)GasP. Alternatively, we may want to restrict the number of channels per module or module type, based performance targets for area, latency, and throughput.

- Click modules use *standard-cell* Verilog implementations. The module implementations are programmed and stored in a database in the compiler, and used by the Parser to generate a Verilog netlist for a given GUI design. The Verilog descriptions of the standard cells are provided through a separate library. (T)GasP assumes *custom-cell* implementations, possibly supported with a corresponding layout, simulated extensively in SPICE, and designed using Electric [48]. The Verilog implementations stored in the compiler database for (T)GasP are imported from Electric.

3.3.3 Static Validation Changes

ARCwelder offers a modular approach to static timing analysis. Each module comes with a set of timing constraints. The constraints vary from manufacturing

constraints such as minimal clock pulse widths for flipflops, bundled data constraints such as data setup and hold constraints, and control constraints. The manufacturing constraints manage lower level analog and silicon implementation aspects. The bundled data constraints manage correct handover of data through the communication channels. The control constraints manage the stabilization of the self-timed logic between subsequent communications.

The constraints are formulated and validated on a module-by-module basis, but their computations may span several modules. The handshake channels act as the timing interface between the modules. The full expansion of the constraints is done during validation. Where expansion is needed, the timing constraints in the modules are formulated by induction, assuming that necessary timing information will be handed over through the handshake channels. This works beautifully, and allows for the flexible skewing of data versus control that's often needed to optimize the circuit performance in speed or power. Willem Mallon refers to this as *Bounded Bundled Data* [18].

The timing constraints are stored in a special file, called "HC Timing Constraints" in Figure 3.1.1. The constraints express the relation between certain computation paths — e.g. how the delay through one path must be shorter than the delay through other paths, starting from the same location in the circuit. With each constraint, also a repair location is given, i.e., a location in the Verilog netlist where extra delay can be inserted to increase the delay of the intended longer path so it's indeed longer than that of the intended shorter path. For more details on the semi-automatic generation of such timing constraints, see the PhD work by Hoon Park [39, 41].

We can use the static timing analysis code in ARCwelder in three ways:

1. **To validate and correct circuit timing:**

   ARCwelder reads the timing constraints for each module. It computes the delay of the two constrained paths, using the gate delay information and the gate input to output connectivity in the Verilog netlist, and it outputs the difference of the supposedly longer minus the supposedly shorter path in a timing report. A special procedure, called "Repair Timing Failures" in Figure 3.1.1, reads the timing report and updates the netlist by inserting adequate delay at specified repair locations. ARCwelder repeats the whole process for the updated netlist, and its updated netlist, etcetera, until all violations are resolved.

   If the repeated process fails to terminate, i.e., each repetition sees a timing violation, this would be an indication that the timing constraints are inconsistent or incomplete.

2. **To validate the soundness and completeness of the timing constraints:**

   ARCwelder can insert random delays at random locations in the Verilog netlist, and then follow the process in item 1 above to adjust the circuit timing until all timing constraints are satisfied. If this process fails to terminate within reasonable time, then we may assume that the constraints are incomplete or inconsistent. In that case, we investigate the timing reports to understand what to fix where. For instance, the constraints and their repair delay insertion points may have circular dependencies.

   If the process terminates, we run a dynamic validation test for the circuit, to check the handshake behavior and the handover of data. If this test fails, then we may assume that the constraints are incomplete or inconsistent. We investigate the failing run information to understand what's missing or wrong and how to fix it.

3. **To interface with commercial static timing analysis tools:**

ARCwelder can translate the constraints stored in the “HC Timing Constraints” files for the handshake components, and reformulate these so they become readable by standard Static Timing Analysis tools, such as Synopsys PrimeTime. We can then let the standard tools do the analysis.

There are many issues with using standard tools for static timing analysis of self-timed circuits, especially when the circuits have combinational loops — as do GasP and TGasP. I have investigated these issues as part of my MSc research. My MSc thesis explains how one can formulate timing constraints for GasP modules and analyze them using Synopsys PrimeTime [22].

The beauty of the ARCwelder code is that it can pre-partition the statically timed paths into portions that the standard tools can handle, and manage the non-standard loop unrolling inside ARCwelder.

Below follows a list of changes that I applied to make the ARCwelder static timing analysis code work for (T)GasP, with conclusions that I have drawn from these changes.

1. **Separate timing constraint files for (T)GasP:**

   I added a completely new directory with timing constraint files for (T)GasP modules. This is probably unavoidable because the module implementations are different from those in Click. Nevertheless, the code itself could benefit from a higher level of abstraction, so modules with similar timing constraints can share procedures rather than store separate but similar code fragments.

   Seeking such a higher-level of abstraction has become part of Hoon Park’s research. In collaboration with Anping He, Professor Song, and Marly Roncken, Hoon has been identifying the constraint patterns and how they relate to the design patterns of Click and GasP modules [41].

2. **Support for asymmetric delay insertion:**
   Some of the (T)GasP repair solutions use asymmetric delay insertions to maintain a 5-gate-delay drive pulse on the statewire. The asymmetry complicates the formulation of the timing constraints, because the original ARCwelder code is ignorant of the high or low state of a signal when it follows the timing paths from gate to gate. By partitioning the (T)GasP timing constraints at delay insertion points it becomes possible to guarantee that the correct gate input-to-output delay is used, without explicit reference to high or low signal values. This approach, which I used in my ARCwelder code extension, has been subsequently adopted and formalized in the research work of Hoon Park [39, 41].

3. **Termination guarantees for evaluating paths with bi-directional wires:**
   The bidirectional nature of the (T)GasP statewires caused longest path computations to loop forever in ARCwelder. I worked with Willem Mallon to split off a separate code fragment for calculating the longest path in (T)GasP designs. Though this separate code fragment avoids the non-terminating loops, it would be preferable to have a more integral solution that works for Click as well as for (T)GasP. In general, I see both the need and the possibility for a more uniform code base for path tracing, using a predefined collection of start, stop, and partition points, such as flipflops, channel interface signals, and delay insertion points.

   The Verilog netlist analyzed by the ARCwelder static timing analysis code is not the same as the Verilog netlist simulated during dynamic validation. The existing static analysis code works with gate delays and gate connectivity. Its gate connectivity information is unidirectional. Bidirectional transistor-level modules, such as the statewire driver modules in (T)GasP and their half-keepers, or even the transistor models generated by Electric, do not fit in. To

support static timing analysis for TGasP in ARCwelder, I decided on a case-by-case basis which parts of the Verilog netlist to unroll and where to stop such unrolling to substitute strictly unidirectional input-output "gates." This works, but requires additional verification and automation.

3.3.4 Dynamic Validation Changes

ARCwelder supports dynamic validation of any Verilog netlist generated by ARCwelder, be it generated by the Parser or during Static Timing Validation. Before we simulate an ARCwelder generated Verilog netlist, we write a test bench, in Verilog, with additional environment modules that automatically control the netlist inputs and automatically inspect the netlist output values against the values that we expect the design to produce in response to the given inputs. In addition to inspecting a textual test report, noting correct as well as incorrect results, we can inspect graphical waveforms for signals in the design and in the test environment. The ARCwelder code for dynamic validation can support on-the-fly checks to validate proper handshake behavior. For instance, the code can verify that the data sent over the channel are stable during each handshake, i.e., from the handshake's request to its acknowledge signal.

Below I list my key code changes to make ARCwelder's dynamic analysis work for (T)GasP, and conclusions and recommendations that I derived from these.

1. **Handshake checks under different channel initializations**
   I adapted the monitors that check for proper handshake behavior. In Click, all handshake channels are initialized low. In (T)GasP, we can initialize them low (empty) or high (full). I already hinted at this in Section 3.3.2 when discussing the ARCwelder Parser. The original code counts the number of handshake

events per channel, starting at 0. Each monitor assumes that an even number means that all handshakes have completed, i.e., the channel is empty, while an odd number means that the channel is in a handshake, i.e., the channel is full. By initializing the event count of initially high handshake channels to 1, the monitors are in tune again with ARCwelder's odd-even count mechanism of handshake events.

2. **Additional Verilog models for (T)GasP**

   A lot of time went into the development of Verilog models for (T)GasP for our Verilog simulator of choice. Willem Mallon wanted to use Icarus Verilog because no license is needed to run it and because it runs on a laptop. Icarus Verilog works fine for Click but not for (T)GasP circuits. For instance, it doesn't handle the bidirectional transistor models generated by Electric. A dedicated ARC report summarizes my Icarus Verilog models for (T)GasP [24]. The major disadvantage of committing to Icarus Verilog is that the resulting (T)GasP models are too far removed from the Electric models to sign-off on real silicon validation.

   For the Weaver chip [56, 46], a GasP test chip on real silicon, we ignored Icarus Verilog and instead used Mentor Graphics Modelsim for our Verilog simulations. The extra modeling effort in ModelSim concerned only a few constructs — those where analog meets digital, such as arbiters and such as tristate structures where the delays don't quite match and for which Verilog would create X-s while in reality the output state is well-defined. A summary of these Modelsim models can be found in the 2014 ARC report, ARC2014-smg03 [35].

3.4 ARCwelder Click versus (T)GasP Experiments

The new ARCwelder code successfully compiles dataflow designs to Click and to (T)GasP.

To test and compare ARCwelder compiled Click and (T)GasP designs, I experimented with three algorithmic designs: a Fibonacci number generator, a Greatest Common Divisor, and Willem Mallon's LEETL microcontroller [19]. This section contains the details of the Fibonacci experiment [29]. Information about the two other experiments can be found in the 2013 ARC reports, ARC2013-smg04 and ARC2013-smg05 [30, 31].

The Fibonacci design generates a sequence of Fibonacci numbers. More specifically, it generates the sum of the last two numbers in a sequence, primed with 0 followed by 1, thus generating sequence 1, 2, 3, 5, etcetera. Figure 3.4.1 shows my GUI design for the Fibonacci experiments reported in this section. Note that it is different from the GUI's Fibonacci design in Figure 3.3.1. Figure 3.4.1 contains two fullStore modules, *lastFib* and *next_to_lastFib*, one Store module, *temp*, two Fork modules, *f1* and *f2*, and one Join module, *sum*. Each fullStore module starts with valid data, Store starts empty with irrelevant data. The data in *next_to_lastFib* and *lastFib* are structured as 16-bit words and initialized to a value of 0 respectively 1. Channels *con0* and *con3* are the first to handshake. The testbench collects the Fibonacci sequence from *con5* in *tb_out*.

For our experiments, we map the Fibonacci design in Figure 3.4.1 to Click respectively (T)GasP circuits, and we simulate each mapped circuit with gate-level timing information.

We assume a unit-level (inverting) gate delay model of 100ps. We assume that

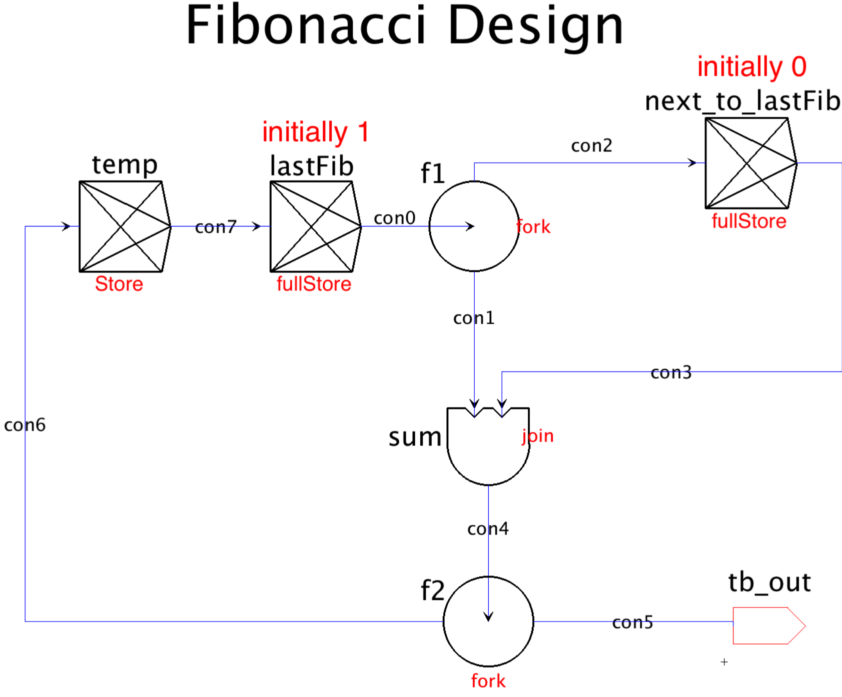


Fig. 3.4.1: Another GUI design for a Fibonacci number generator, generating the sum of the last two numbers in a sequence, primed here with first 0 and then 1, and thus generating 1, 2, 3, 5, etcetera. The design contains two fullStore modules, *lastFib* and *next_to_lastFib*, one Store module, *temp*, two Fork modules, *f1* and *f2*, and one Join module, *sum*. Each fullStore module starts with a data value: we start *next_to_lastFib* with 0 and *lastFib* with 1.

it takes 2 unit-level gate delays to go through an XOR or XNOR gate and 1 unit-level gate delay to flip the output of flipflop *FF* after *ck* goes high in Figure 3.2.1(left).

Under these assumptions, the GasP Store implementation in Figure 3.2.1 has a forward path delay of 600ps from *in[sw]* high to *out[sw]* high, and a reverse path delay of 400ps from *out[sw]* low to *in[sw]* low. Likewise, the Click Store implementation has a forward path delay of 500ps and a reverse path delay of 500ps.

Table 3.4.1 summarizes the corresponding forward and reverse path delays for

Table 3.4.1: Forward and reverse path delays for Click and (T)GasP modules, using 100ps unit-level gate delays, 2 unit-level gate delays for X(N)OR, and 1 for flipflop *FF ck* to *Q*.

	Store	fullStore	Join	Fork
GasP Forward Path Delay	600	600	200	200
GasP Reverse Path Delay	400	400	200	200
Click Forward Path Delay	500	800	500	0
Click Reverse Path Delay	500	500	0	500

Click and (T)GasP implementations of the various modules in the Fibonacci design of Figure 3.4.1.

Table 3.4.2 and Table 3.4.3 show cumulative timing details for generating the first two Fibonacci numbers in the (T)GasP respectively Click implementation of Figure 3.4.1.

The timing details in Tables 3.4.2–3.4.3 indicate that the respective cycle times between two successive Fibonacci numbers observed at *tb_out* are:

- for GasP: 1100 (unfold) + 500 (interfacing) + 1000 (fold) = 2600 ps, and
- for Click: 1300 (unfold) + 500 (interfacing) + 1500 (fold) = 3300 ps.

So, the GasP implementation has a 21% faster cycle time than the Click implementation.

In other words: despite the absence of a fast reset signal in the TGasP implementations used in this experiment, the timing results of the Fibonacci implementation with (T)GasP modules are better than the timing results for the implementation with Click modules. This is also true for the two other experi-

ments [30, 31]. Perhaps, the unavoidable reverse latency for single-track telescope solutions, like TGasP, is less of a disadvantage than suggested in Chapter 2.

Specifically — without delay for data computations — we observed that:

- for the Fibonacci design, (T)GasP has 21% higher throughput than Click,
- for the Greatest Common Divisor, (T)GasP has 34% higher throughput than Click,
- for LEETL, GasP has 24% higher throughput than Click.

A larger delay for data computations, e.g., for $+$ in module *sum*, will bring the throughput results for the Click implementations closer to those for (T)GasP.

Table 3.4.2: GasP timing information for the first two Fibonacci results for Figure 3.4.1.

Path and Delay Information	**Cumulative Delay [ps]**
Initialization **makes** 2000ps	2000
First telescope unfolding: *f1*(200ps) + *sum*(200ps) + *f2*(200ps) **makes** 600ps	2600
tb_out-handshake(500ps) **in parallel with** *con6*-handshake(500ps) **makes** 500ps	3100
First telescope folding: *f2*(200ps) + *sum*(200ps) + *next_to_lastFib*(400ps) + *f1*(200ps) **in parallel with** *temp*-finish-copy(100ps) **makes** 1000ps	4100
Second telescope unfolding: *con0*-handshake(500ps) + *f1*(200ps) + *sum*(200ps) + *f2*(200ps) **makes** 1100ps	5200

Table 3.4.3: Click timing information for the first two Fibonacci results for Figure 3.4.1.

Path and Delay Information	**Cumulative Delay [ps]**
Initialization **makes** 2000ps	2000
First telescope unfolding: *lastFib*-start(200ps) + *f1*(0ps) + *sum*(500ps) + *f2*(0ps) **makes** 700ps	2700
tb_out-handshake(500ps) **in parallel with** *con6*-handshake(500ps) **makes** 500ps	3200
First telescope folding: *f2*(500ps) + *sum*(0ps) + *next_to_lastFib*(500ps) + *f1*(500ps) **in parallel with** *temp*-finish-copy(0ps) **makes** 1500ps	4700
Second telescope unfolding: *con0*-handshake(800ps) + *f1*(0ps) + *sum*(500ps) + *f2*(0ps) **makes** 1300ps	6000

3.5 Summary and Conclusion

This chapter gave an overview of the ARCwelder design flow, and of the changes I made to ARCwelder to compile dataflow designs to Click as well as GasP and Telescope GasP circuits. The chapter ends with a comparison of a Click versus (T)GasP compiled Fibonacci number generator. In addition to the Fibonacci design, I tested and compared Click and (T)GasP implementations for two other designs [30, 31]: a Greatest Common Divisor, and Willem Mallon's LEETL micro-controller [19]. Ignoring the data computation delays, the compiled (T)GasP designs all have a higher throughput than the compiled Click designs. Additional delay for data computations, like the numeric addition used in the Fibonacci design, will bring the throughput results for the Click implementations closer to those for (T)GasP. A summary of how to run such experiments, and a more detailed evaluation of the three Click versus (T)GasP design experiments can be found in separate ARC reports [32, 29, 30, 31].

The resulting ARCwelder compiler for Click and (T)GasP uses a lot of code duplication. The amount of code duplication points to the need for a more integral design flow. My ARCwelder study and experiments have served as an eye opener for how we at the ARC and how the self-timed research community at large have created a "Tower of Babel":

Our "Tower of Babel" — By exposing the differences between our communication protocols to our design and test flows, it has become practically impossible to share tools and designs between different self-timed circuit families.

The next two chapters explain the ARC's new point of view on design and test.

This new point of view exposes the similarities between the various self-timed circuit families, without sacrificing their peculiarities.

3.6 My Contributions for Chapter 3

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

- My ARCwelder compiler study has changed our point of view for design and test at the ARC. The new point of view and related developments were published by Marly Roncken et al. [46], and are described in Chapters 4 and 5 of this thesis.
- I imported the Telescope GasP modules described in Appendices A, B and C into ARCwelder. I modeled the modules in Verilog, using Verilog annotation scripts in Electric – the ARC's design environment for GasP-style circuits. The Verilog models were then stored in the Parser's "HC & Gate Library" — see Figure 3.1.1.
- While importing (T)GasP modules into ARCwelder, I discovered a bug in the master-clear implementation. This bug has two parts. First, during master-clear, all statewires were initialized low (empty). Second, the end of master-clear served as a start signal, thus confounding master-clear and start. These master-clear implementations worked fine for previous GasP modules but are no longer adequate for new GasP modules developed at the ARC for use with ARCwelder. For one, the fullStore module requires a high (full) statewire for its output channel. Second, initialization

takes time, especially in the presence of telescope modules. To give initialization the time it needs, master-clear and start signals must be different.

Note

The master clear bug has been solved in the new design and test approach described in Chapters 4 and 5 of this thesis [46]. The new approach can initialize a statewire high (full) or low (empty), as needed. The new approach also separates initialization from execution — see Section 5.2.1.

4

Naturalized Communication

Both my ARCwelder compiler extension to (T)GasP – see Chapter 3 — and Hoon Park's ARCtimer timing verification flow [39, 41] suffered from seemingly unnecessary code duplication. We had expected to be able to reuse more of the existing compilation solutions for Click when compiling to (T)GasP. But even small differences such as initialization were more deeply embedded in the compiler code than we had anticipated. Likewise, Hoon Park's verification efforts required him to specify and verify communication parts of the design even when only the computation part changed.

Both my and Hoon's research resulted in a new point of view for how to design self-timed VLSI systems. At the ARC, we started to view self-timed circuits as networks of links and joints. Links deal with the communication issues, including data storage, full and empty storage, data transport, and full and empty transport. Joints deal with the computation issues, including flow control, arbitration, and arithmetic operations. Instead of using handshake protocol interfaces, we now use full-empty interfaces. This allows mixing and matching of self-timed circuit families and reduces redundant work in building computer aided design (CAD) support for the various circuit families.

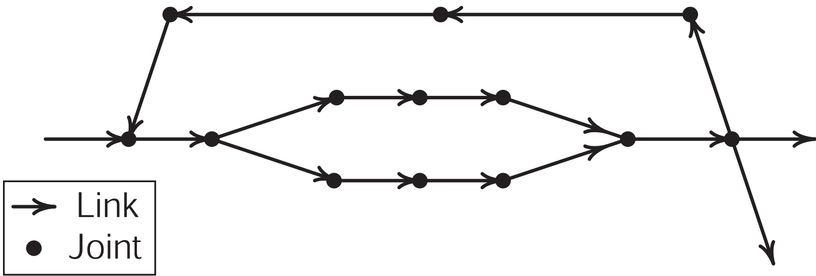


Fig. 4.0.1: A self-timed dataflow system with communication channels, called *links*, and flow control and data computation modules, called *joints*, can be viewed as a directed graph with data flowing in the direction indicated by the arrows.

A separation of communication and computation is beneficial because it allows us to verify the communication parts independently from the computation parts — and vice versa. The full-empty interfaces between the two emphasize the commonality between the different circuit families, including Click and GasP. By moving the interface from the handshake signals — which are different for Click and GasP — to full, empty, make-full (fill), and make-empty (drain) signals — which are the same for Click and GasP — Click circuits can now "talk" directly to GasP circuits.

Figure 4.0.1 shows a self-timed dataflow graph using links as edges and joints as nodes. The links are the self-timed communication channels, with data flowing in the direction of the arrows. The joints are the self-timed computation modules that implement flow control and data operations.

This chapter presents the ARC's novel point of view for designing with such links and joints. This point of view was published and presented at ASYNC 2015 [46]. Appendix D contains a copy of the publication. The presentation can be downloaded from the ARC website [2].

Below, we contrast systems with handshake interfaces (Section 4.1) against

systems with full-empty interfaces (Section 4.2).

4.1 Systems with Handshake Interfaces

We see a self-timed system as a directed graph of links and joints, as shown in Figure 4.0.1. Figure 4.1.1 represents a joint as a stick figure with data flowing in the direction of the arrow, and links as rectangles. Each link-joint-link triple in Figure 4.0.1 represents a First In First Out (FIFO) buffer, with an input link, called *in*, and an output link, called *out*. The most important property of a link is whether it is full or empty, just as the most important property of a parking place is whether or not it is occupied. We color the rectangle of a full link dark and leave an empty link white.

Each link reports its full or empty state at both ends. It accepts a fill command at its input end and a drain command at its output end. Fill and drain commands

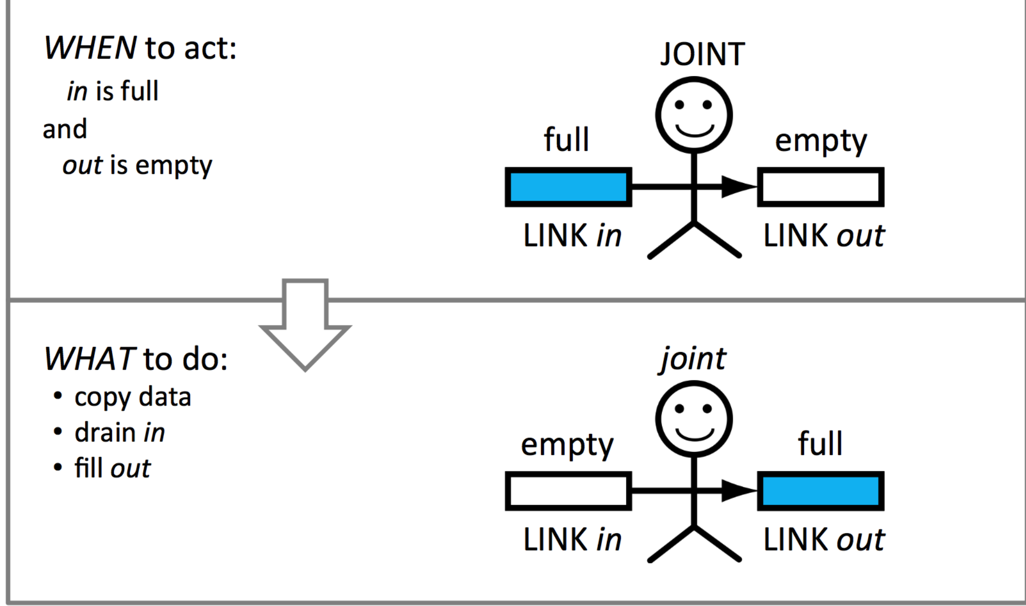


Fig. 4.1.1: A pictorial representation of a self-timed FIFO action.

change the state of a link. The impact of fill and drain commands is observed immediately at the near end of the link but may take time to traverse the length of the link before appearing at the link's far end.

The FIFO in Figure 4.1.1 starts with empty links. When its input link *in* is full and its output link *out* is empty, the joint performs three tasks:

- it captures and hands over the data from link *in* to link *out*,
- it drains link *in*, making it empty, and
- it fills link *out*, making it full.

These tasks are performed in parallel. They are repeated when the joint's input link, *in*, has new data and is again full, and its output link, *out*, has transferred the captured data and is again empty.

The representations for full and empty links depend on the specific handshake protocol. Click uses a non-return-to-zero (non-RTZ) variant and GasP uses a return-to-zero (RTZ) variant. Their full-empty representations are as follows — see also Figure 4.1.2:

- **Full and empty link representations in a non-RTZ variant (Click)**
  This representation uses two boolean- or bit-signaling wires: a *request* signal from the sender to the receiver of the data, and an *acknowledge* signal in the reverse direction. The typical convention is that the link is full when request and acknowledge differ, and the link is empty when the two signal values are the same. Note that the link is full for request-acknowledge values 0-1 (low-high) and 1-0 (high-low). Likewise, the link is empty for values 0-0 (low-low) and 1-1 (high-high).

- **Full and empty link representations in a RTZ variant (GasP)**
  This representation uses a single bi-directional signaling wire, called *statewire*. The typical convention is that the link is full when the statewire is 1 (high), and the link is empty when the statewire is 0 (low).

Figure 4.1.3 shows a FIFO GasP module and a FIFO Click module. The two modules differ in how they represent full and empty links, and how they capture data.

Different handshake variants lead to different self-timed circuit families. Well-known circuit families that use two-phase bundled-data handshake protocols are Micropipeline [51], GasP [55, 53], Mousetrap [49], and Click [43]. Micropipeline, Mousetrap and Click use a non-return-to-zero (non-RTZ) variant. GasP uses a return-to-zero (RTZ) variant.

Figure 4.1.4(top) shows a longer FIFO using one GasP module and one Click module. GasP was invented and developed by researchers at Sun-Oracle Laboratories in the United States of America. To honor these American-based researchers, the GasP module wears a cowboy hat. Click was invented and developed by researchers at Philips Handshake Solutions, in The Netherlands. To honor these Netherlands-based researchers, the Click module wears wooden shoes.

If this FIFO works correctly, it will copy the data from left to right and thus move the blue color of the rectangle, an indicator for the link being “full,” from the left-most link to the right-most link. Figure 4.1.4(bottom) shows that this fails to happen: something goes wrong in the middle link.

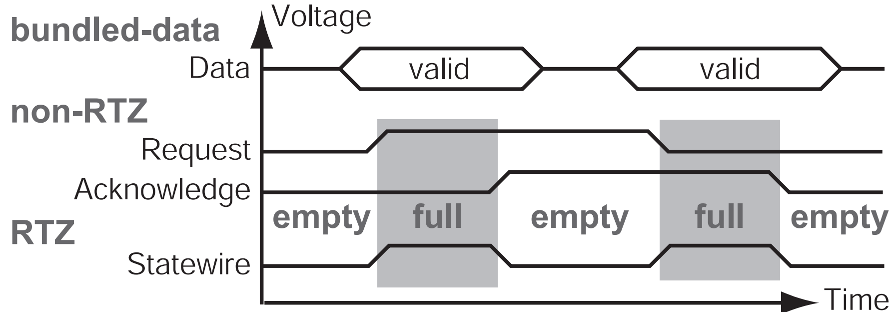


Fig. 4.1.2: Full-empty representations for non-RTZ and RTZ bundled-data handshake protocols. Data are valid when the link is full, and may change only when the link is empty.

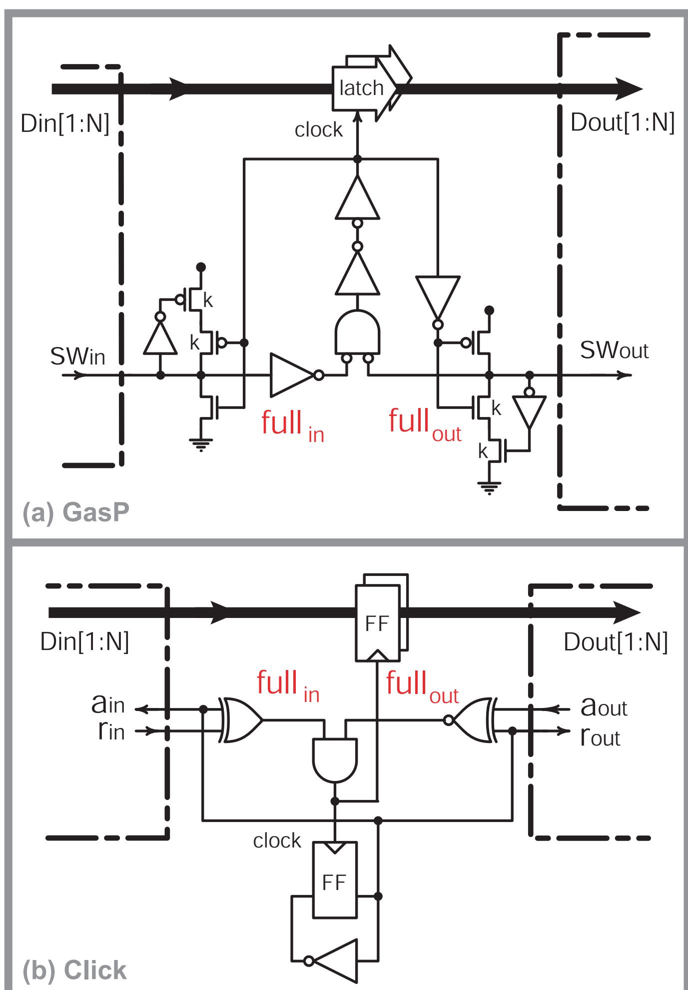


Fig. 4.1.3: Original circuit designs for a single-input-output FIFO module in GasP (a) and Click (b). The dashed lines mark the interface between the joint in the middle and the left-hand and right-hand links. Note that the joints hold all the logic — the links are “just wires.”

Figure 4.1.5(top) zooms into the middle link's handshake interface, to see what went wrong. The data connections match. But there is no direct way to connect the single bidirectional GasP handshake signal to the two unidirectional Click handshake signals so they convey the same full and empty information.

To make this GasP-Click communication happen requires that we add protocol translators in the link. Figure 4.1.5(bottom) shows an example of two translators for filling a link and reporting the link's full state back. Translators are expensive and they cost extra validation effort in addition to extra area, time and power.

The point of this mixed GasP-Click FIFO is to illustrate that handshakes may be fine to implement self-timing, but as *interfaces* they complicate collaboration and design reuse.

There is an alternate solution for designing mixed handshake systems without the need for protocol translators. In Section 4.2, we present a link-aware design approach that de-emphasizes the differences in "how full and empty links are represented" and "how data are captured" by moving both the full-empty logic and the data storage outside the joints and into the links. This results in a standard link-joint interface that works for different self-timed circuit families and that allows translator-free communication between them.

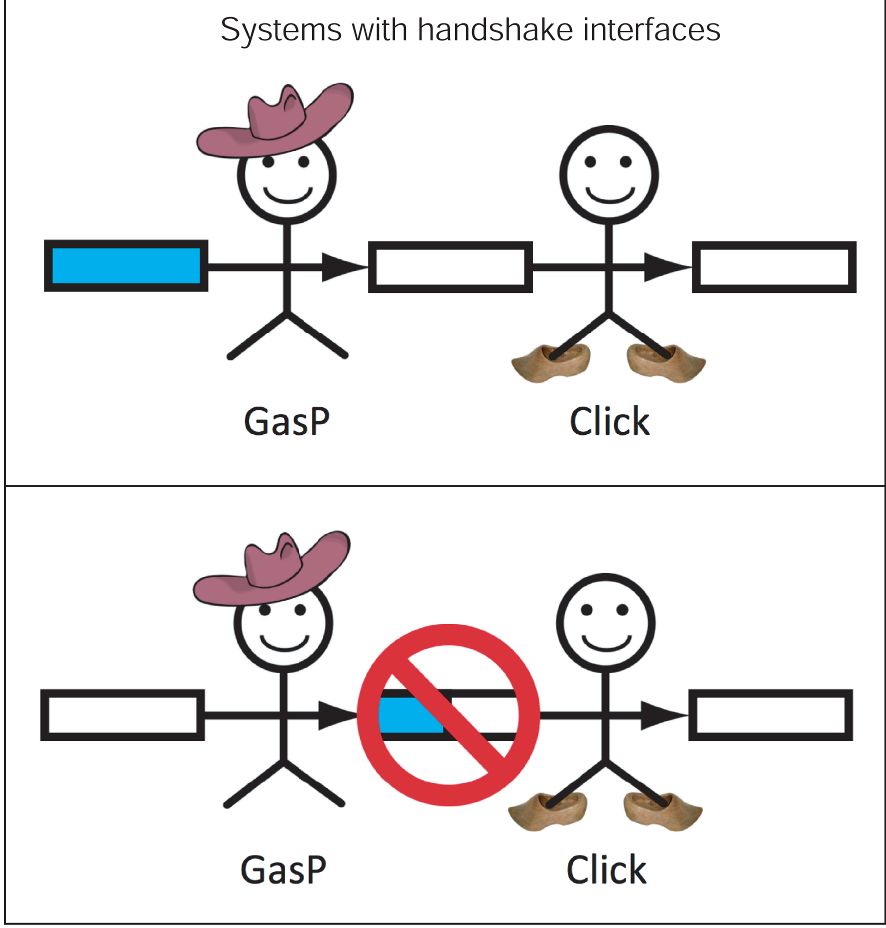


Fig. 4.1.4: (top) Pictorial view of a FIFO with GasP and Click modules, and a full input link. To honor their inventors, the GasP module wears a cowboy hat and the Click module wears wooden shoes. (bottom) The link which serves as a handshake interface for both the GasP module and the Click module fails as a communication interface.

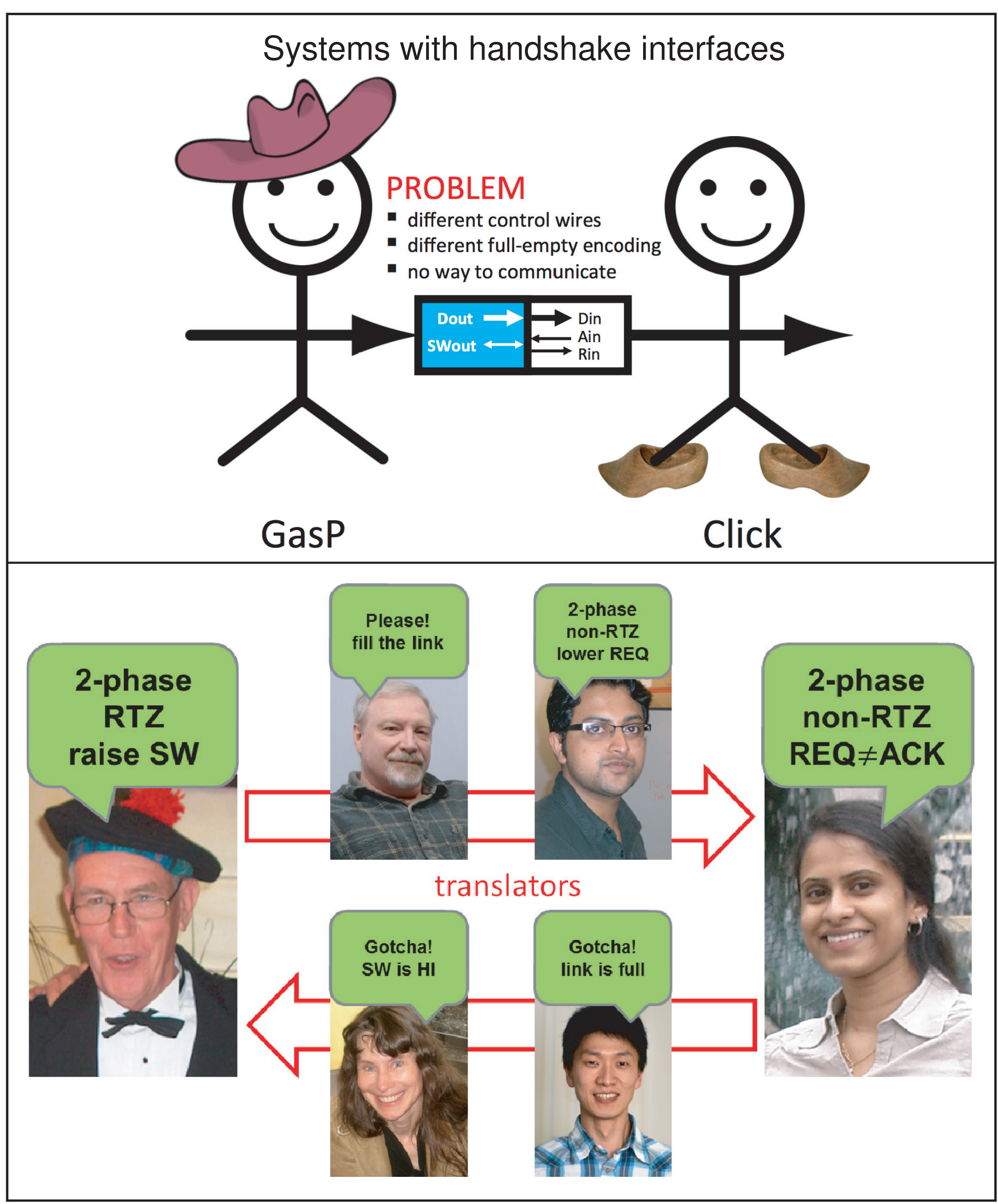
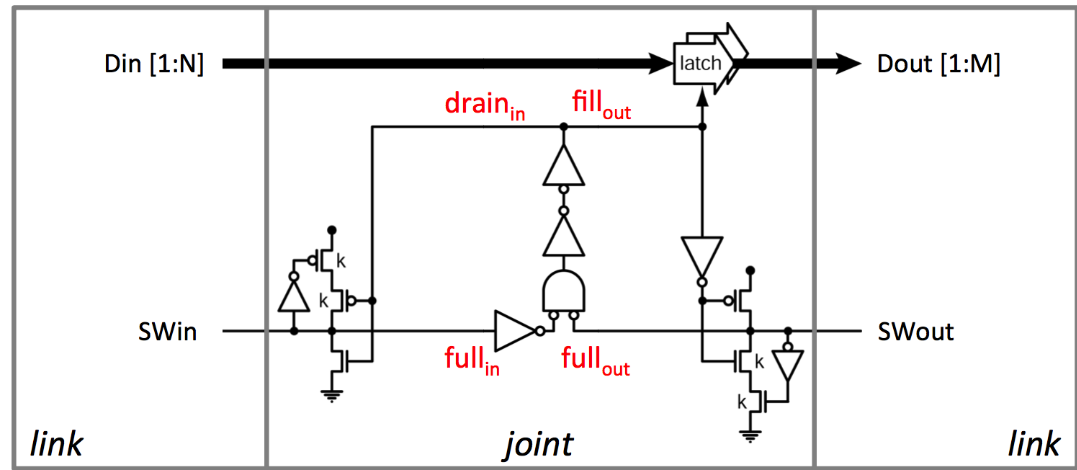


Fig. 4.1.5: (top) Detail of the failing communication over the middle link in Figure 4.1.4. The GasP module on the left uses a single bidirectional statewire and the Click module on the right uses unidirectional request and acknowledge signals to represent “full” or “empty.” There is no way to wire these control signals so they convey the same full-empty information. (bottom) Example of translators to fill a link between different handshake modules, and report the link’s full state back to the sender module.

4.2 Systems with Full-Empty Interfaces

Figure 4.2.1(top) repeats the GasP FIFO module of Figure 4.1.3(a). Figure 4.2.1 (bottom) does the same but also moves the full and empty link retention as well as the data latches into the links. The interface signals thus created sense the full-empty state of the links ($full_{in}$, $full_{out}$), handover data (D_{in}, D_{out}), make an incoming link empty ($drain_{in}$), and make an outgoing link full ($fill_{out}$). Inside each GasP link one now sees data latches, a bidirectional statewire, half-keepers, and pull-up and pull-down drivers. Outside, one sees D, and unidirectional full, fill, and drain signals.

Figure 4.2.2(top) repeats the Click FIFO module of Figure 4.1.3(b). The bottom figure does the same but also moves the full and empty link retention as well as the data flipflops into the links. The interface signals thus created sense the full-empty state of the links ($full_{in}$, $full_{out}$), handover data (D_{in}, D_{out}), make an incoming link empty ($drain_{in}$), and make an outgoing link full ($fill_{out}$). Inside each Click link one now sees data flipflops, a request signal, *R*, an acknowledge signal, *A*, and XOR and XNOR gates that translate R and A to full and empty. Outside, one sees D, fill, drain, and full. We also duplicated control flipflop *FF*, moving one copy into the left link and the other copy into the right link. This solution is better than the original Click FIFO solution in terms of modularity and storage capacity when used in larger systems – see Appendix D for details.



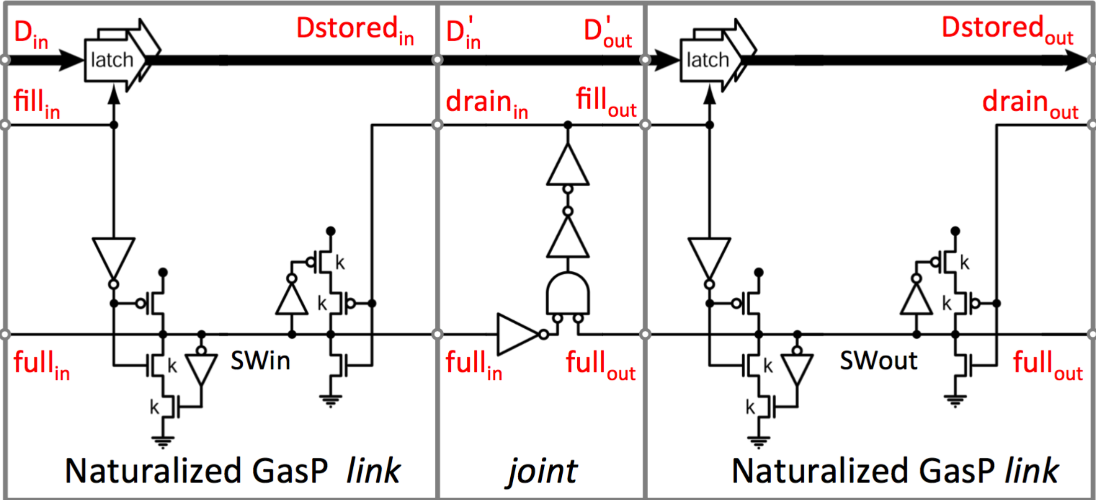
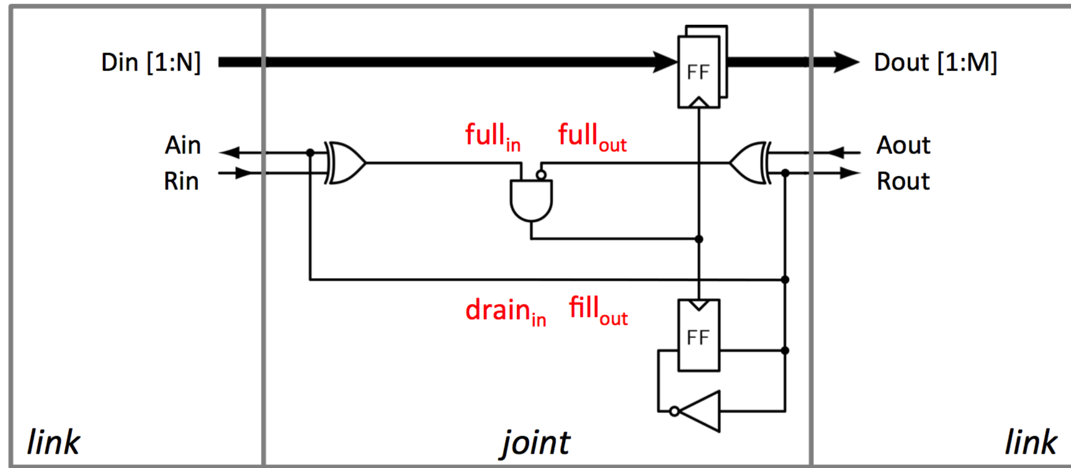


Fig. 4.2.1: GasP circuit with handshake Interfaces (top), and full-empty interfaces (bottom).



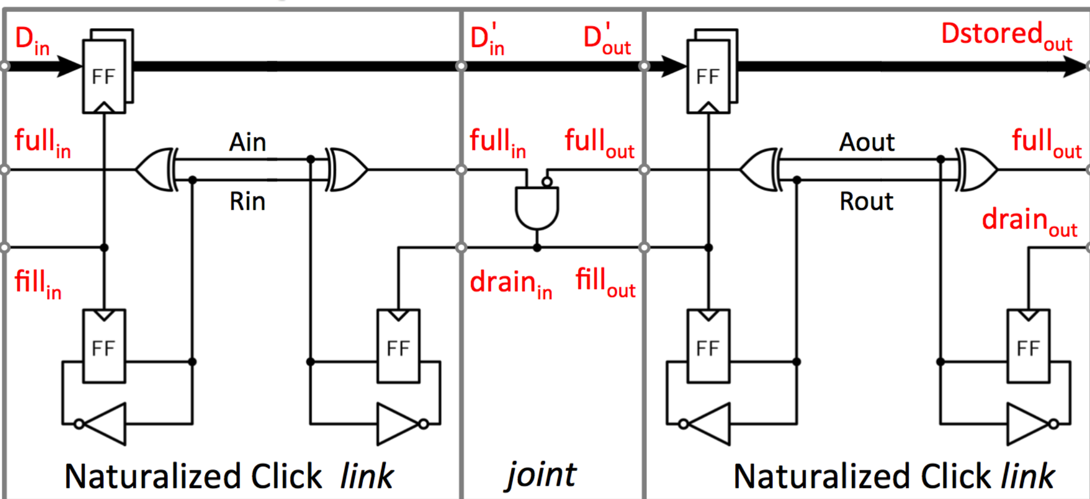


Fig. 4.2.2: Click circuit with handshake Interfaces (top), and full-empty interfaces (bottom).

When we repeat the mixed FIFO experiment of Figure 4.2.3(top) using GasP and Click links and joints with full-empty interfaces, all goes well — see Figure 4.2.3.

Note that links as well as joints in Figure 4.2.3(top) are marked as GasP or Click. This makes sense for the links because of the handshakes inside them. But FIFO joints with full-empty interfaces are in essence the same for GasP and Click. We call the resulting links and joints "naturalized" links and joints, as explained in the following section.

4.2.1 Naturalized Links and Joints

Our link-aware design view puts equal emphasis on links and joints, by giving each the digital logic needed to perform its role in the system. The term "naturalized" comes from the idea that naturalized citizens share the same rights as native-born citizens. We naturalize links, giving them status equal to joints.

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. The two links in Figure 4.2.1 (bottom) are examples of a naturalized link implemented in GasP. The two links in Figure 4.2.2 (bottom) are examples of a naturalized link implemented in Click.

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.

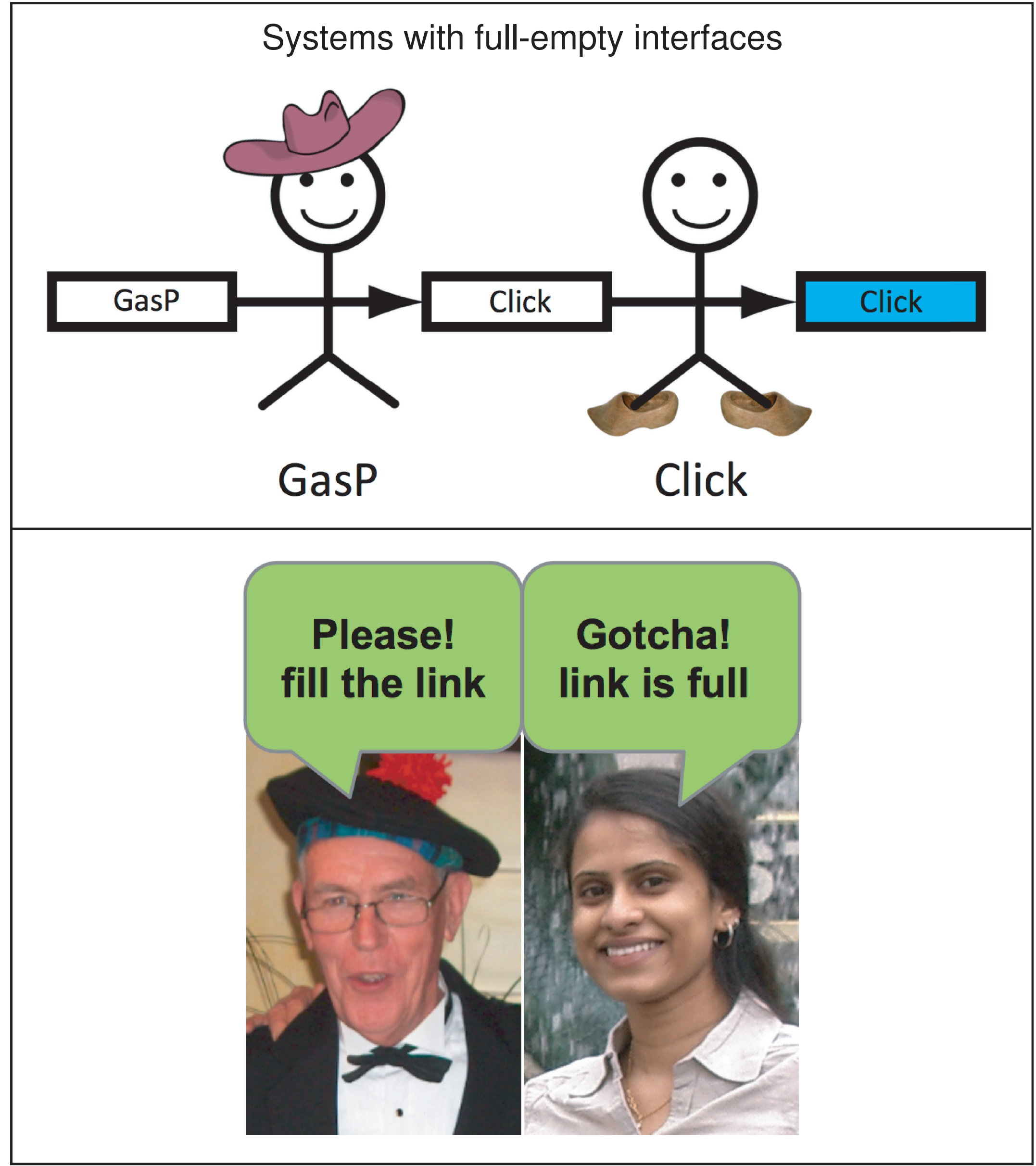


Fig. 4.2.3: (top) The mixed FIFO of Figure 4.1.4–4.1.5 with one full input link works correctly when we use full-empty interfaces. The final state shown here indicates that the data and blue color of the rectangle were handed over correctly from left-most to right-most link. (bottom) Mixed systems with full-empty interfaces communicate without translators.

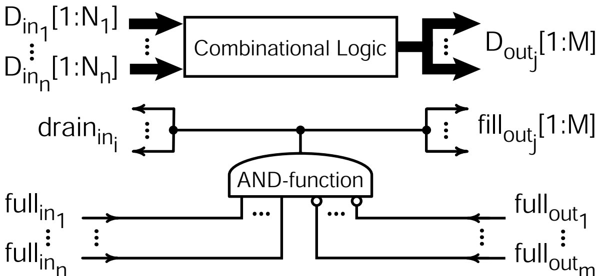


Fig. 4.2.4: Design sketch of a joint without stored state with n naturalized incoming and m naturalized outgoing links, where $1 \leq i \leq n$ and $1 \leq j \leq m$. If data are just copied then $D_{out_j}[1:M]$ is the concatenation of $D_{in_1}[1:N_1]$ to $D_{in_n}[1:N_n]$ and $M = N_1 + \ldots + N_n$ $(N_1, \ldots, N_n \geq 0)$.

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 actions may persist for only a short time, a time whose duration depends on the circuits in the feedback loop.

Joints that pipeline, fork, or join combinational dataflow operations can be free of stored state. Figure 4.2.4 sketches the design of such a joint. The joints in Figure 4.2.1 (bottom) and Figure 4.2.2 (bottom) are examples of such a joint — with *n=m=1*.

The store-free joint shown in Figure 4.2.4 works with any of the naturalized links. 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. 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 joints significantly easier to understand and design. More details about naturalized links and joints can be found in Appendix D.

4.3 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 in Appendix D. System designers can choose which type to use based on system demands for power conservation or data latency.

Figure 4.3.1 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 [43] 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.

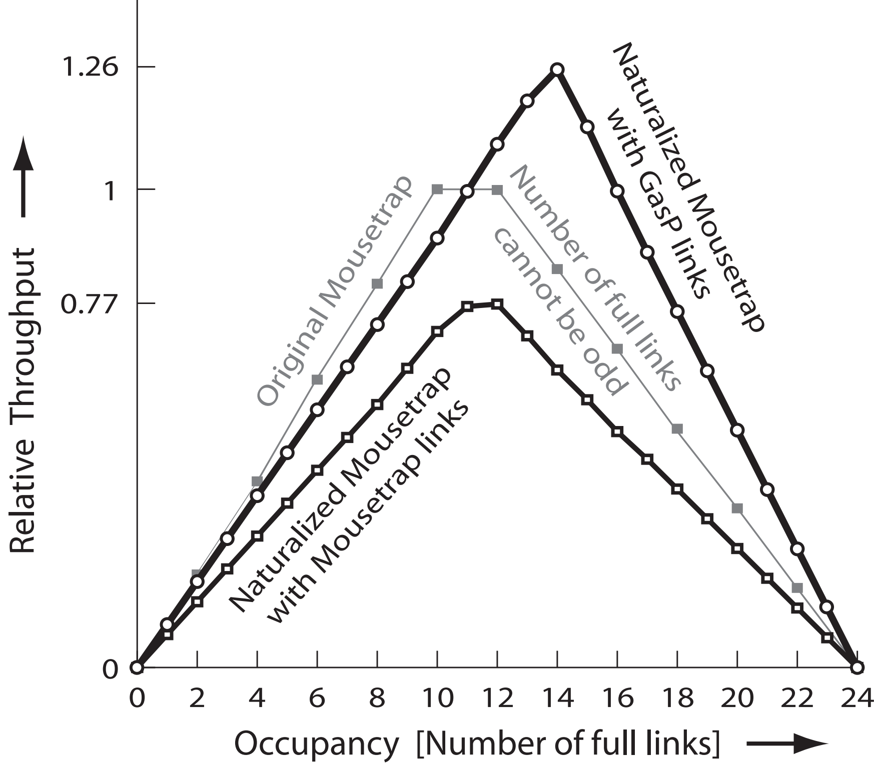


Fig. 4.3.1: Three canopy graphs for simulations of rings with 24 pipeline stages. The center graph with fewer data points is from simulation with an original Mousetrap module. Naturalizing its links with the circuit of Figure 6(c) in Appendix D produces the lower graph. The increased number of logic gates costs performance. Using the link circuits of Figure 6(b) in Appendix D produces the upper graph, improving the original performance. These three 90nm simulations omit data latches and wire loads.

4.4 Summary and Conclusion

This chapter is built around a novel point of view to "*naturalize*" the communication links, giving them status equal to computation joints [46]. *Naturalized communication* unifies thinking about a wide variety of self-timed circuit families and facilitates mixing and matching multiple self-timed circuit families within a single system.

Differentiating links from joints is a simple idea of great power because it offers a higher level of abstraction for each. The simple full-empty interface between links and joints and the resulting unification of the Click and GasP 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.

The next chapter shows how this novel point of view also empowers us to test, debug and characterize self-timed circuits and systems.

4.5 My Contributions for Chapter 4

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

Naturalized communication unifies thinking about a wide variety of self-timed circuit families. Though not directly my discovery, the “naturalization communication” concept has been influenced greatly by the results of my PhD research on the ARCwelder compiler and by Hoon Park’s PhD research on ARCtimer, the ARCwelder timing verification flow.

Hoon and I worked with sufficiently different and yet also sufficiently similar circuit families — Click and GasP — to expose a lack of reuse in the respective compiler and verification flows that we worked with. Both our PhD studies exposed a lack of reuse of similar results between the two circuit families as well as a lack of reuse of similar results between different modules within a family This acted as an eye opener to our supervisors. In her ASYNC 2015 presentation of the Naturalized Communication and Testing paper[1], Marly started by saying [2, page 1 of presentation notes]:

> *Naturalized communication and testing has been an amazing team builder.*
> *The ideas came out of the research work by the first three authors*
> *and have been tested on silicon by the other three authors.*

Marly, Hoon and I are the first three authors, and Navaneeth, Chris and Ivan are the other three authors.

[1] M. Roncken, S. Mettala Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland, “Naturalized Communication and Testing” In *P*roc. IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pages 77–84, May 2015.

5

Naturalized Testing

In the context of this chapter "testing" means validating that the fabricated design-on-silicon operates as intended [6, 8]. This includes *structural testing*, by which we mean low-speed testing to uncover fabrication defects, as well as *functional testing* and *at-speed testing* to uncover incorrect or marginal functionality.

This chapter is based on the ARC's ASYNC 2015 paper [46], shown in Appendix D, and on the corresponding presentation on our ARC website (http://arc.cecs.pdx.edu/) [2, 1].

5.1 Testing: Where we Came from, Are, and Go

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 often reuse the existing clock to start and stop the action under test [61]. This works because each cycle in the design contains a clocked state-holding element. The "tick" governs every synchronous loop.

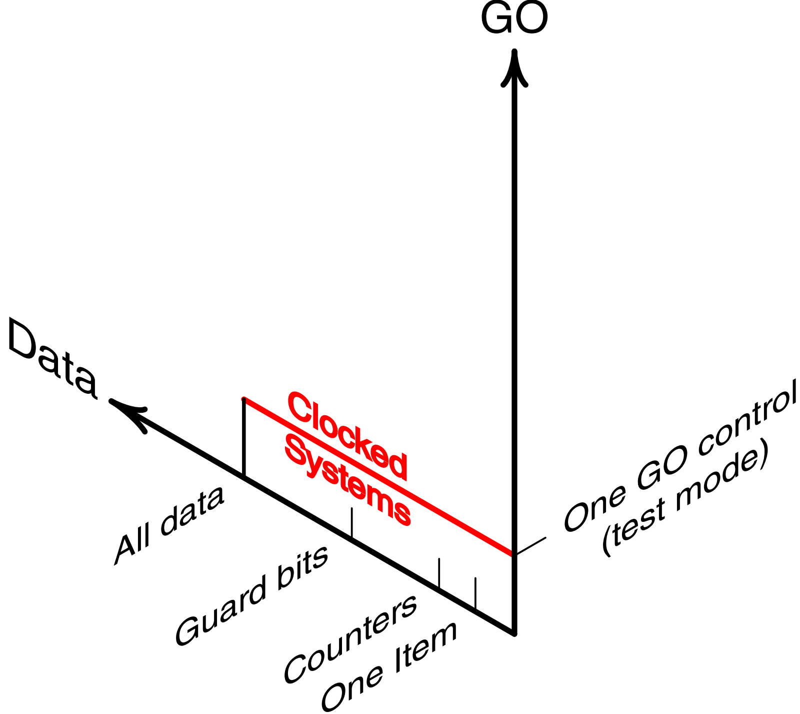


Fig. 5.1.1: Where we came from: synchronous test view. The vertical axis controls the clock, by enabling or disabling it. We call it *GO control* – you may call it *test mode*. Traditional clocked systems have one GO control. The horizontal axis labeled *Data* shows which part of the global state is scanned. If only some data items are scanned, say just the counters, it's a *partial* scan test solution. If all data are scanned, it's a *full* scan test solution.

Figure 5.1.1 shows a two-dimensional image of the test solution used for synchronous systems. The vertical axis, labeled *GO*, controls the clock by enabling and disabling it. Traditional clocked systems have one GO control, also known as *test mode*. The red line indicates which part of the state is scanned. The test solutions represented by the red line go from *no scan* to *partial scan* to *full scan*.

Even though we use two axes to describe this red line the line itself is one-dimensional. From our point of view, traditional scan test is a one-dimensional test method.

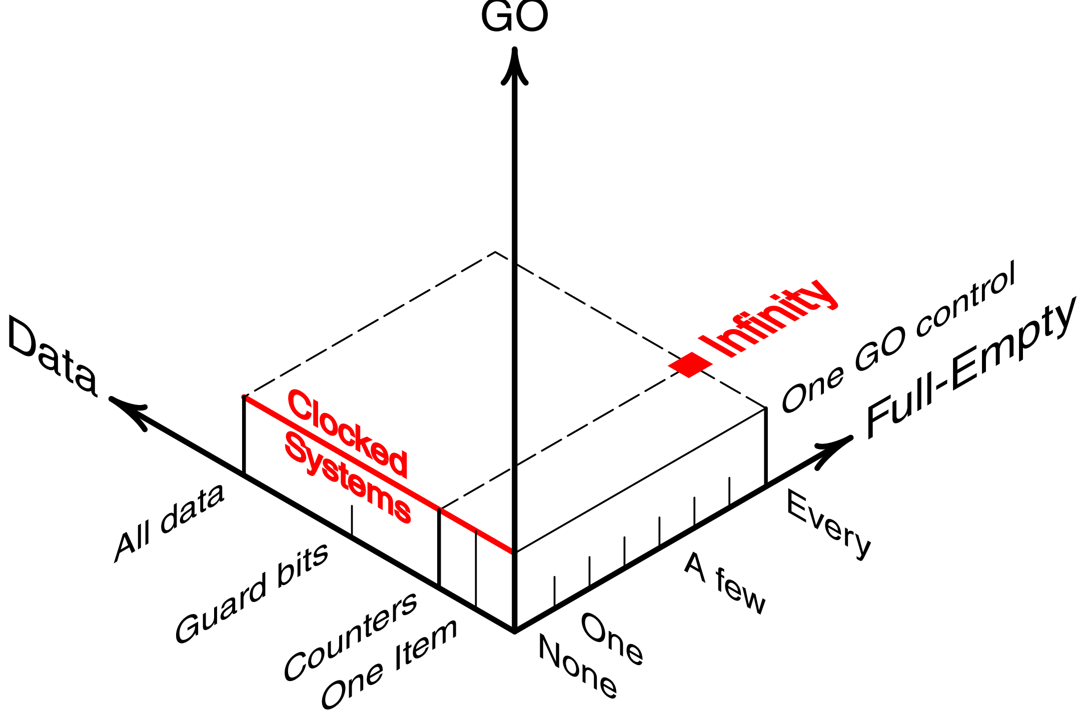


Fig. 5.1.2: Where we are: synchronous test view applied to self-timed systems.

The test solution becomes two-dimensional when we apply it to self-timed systems. Figure 5.1.2 shows not a two- but a three-dimensional image of this test solution applied to self-timed systems. The additional third axis on the right-hand side of the image is labeled *Full-Empty* and represents the control states of the links. The Infinity chip designed by Sun Microsystems in 2008:

- scans every full-empty state,
- scans the counters,
- and can load and unload one Data item.

With exactly one GO control, Figure 5.1.2 shows really a two-dimensional plane with test solutions. We went from one-dimensional — the red line — to two-

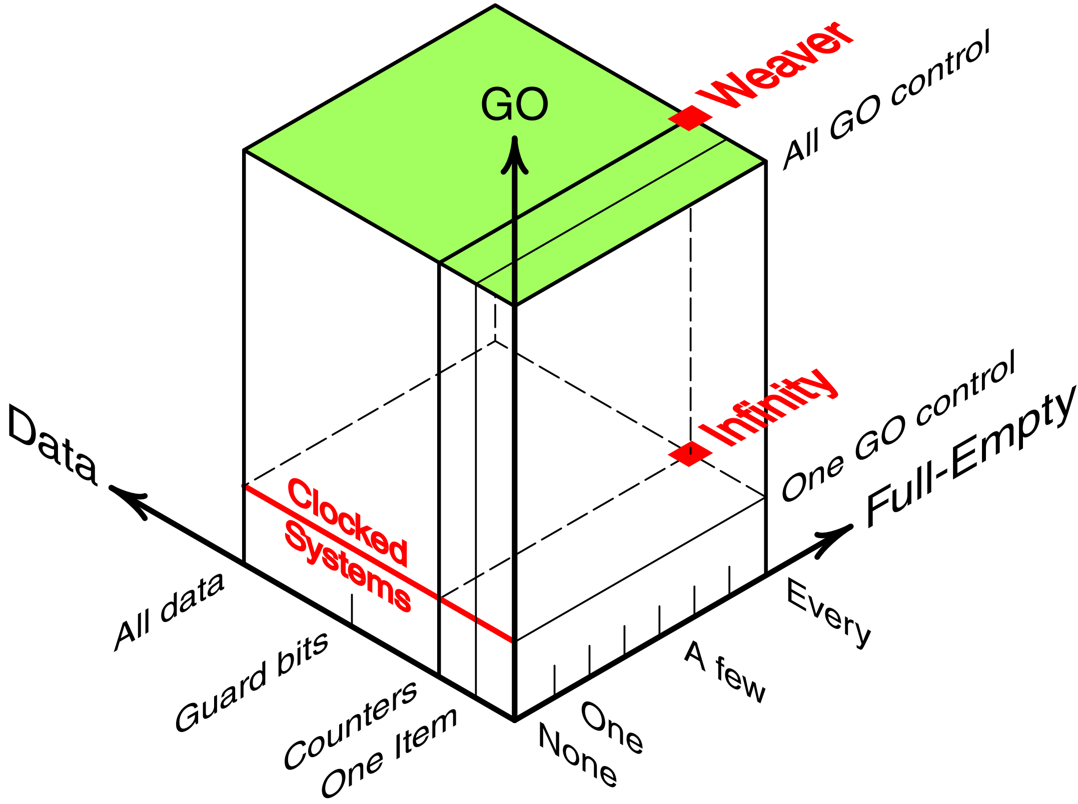


Fig. 5.1.3: Where we go: three-dimensional test view of self-timed systems.

dimensional.

To see the third dimension requires that we recognize that a self-timed system isn't about global action. The actions of a self-timed system are spontaneous, self-generated, and widely distributed in both space and time. The ability to control these distributed actions *separately* matters. That's what the GO axis is for, to control one action, or two, or three, . . . or all of them. Our latest chip, the Weaver scans all GO control signals, every full-empty state, and the counters. In addition it can load and unload one Data item.

What you get with this dedicated GO action-control goes way beyond stuck-at fault detection. Dedicated action control combined with traditional scan access

to Data and Full-Empty states gives you not only stuck-at fault detection, but also at-speed test, debug, and characterization. Examples follow in Section 5.3.

5.2 Naturalized Testing

What makes self-timed circuits "tick"? One may be tempted to point at the locally generated clocks, but these are just by-products of the data. 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 actions.

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 the two in order to reduce test access costs or to fine-tune or reuse test solutions for debug. In this chapter, we call attention to actions, and let them play their own part in the test solution.

The term "naturalized" in the title of this section and chapter comes from the idea that naturalized citizens share the same rights as native-born citizens. We naturalize actions, giving them status equal to states.

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.

5.2.1 Takeoff: From Initialization to Self-Timed Operation

Because each link stores its own full-empty state, links require initialization. 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 4.2.4. 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. We can make each test insensitive to delay variation in different *go* signals.

We can deliver initialization signals via a scan chain. A single global *go* signal would suffice to freeze the system for initialization.

5.2.2 Landing: Stopping a Self-Timed Operation in Full Flight

Molnar et al. recognized long ago how to stop a self-timed circuit using a *proper stopper* [37]. 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* [20].

Once stopped, it is useful to sense the state of the system. The same scan chain that provides initial values — see Section 5.2.1 — can sense the state of the links.

When used to re-initialize the system, for instance for the purpose of priming the next test, a proper stopper must be added *after* the AND function of a joint's action [46].

5.2.3 MrGO

We have found it convenient to combine the *go* signal and the arbitrated stop in one circuit: MrGO, pronounced "Mister GO" — see Figure 5.2.1(top)[1]. When appended to the AND-functions of the joints, as in Figure 5.2.1(bottom), it serves as proper starter for takeoffs and as proper stopper for landings. MrGO also helps us test the circuit, as we will show in Section 5.3.

5.3 Testing with MrGO

For test control, it is essential that MrGO be placed inside the non-blocking 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 action individually, each MrGO gets a separate *go* signal from a scan chain.

[1] Marly Roncken discovered the need for individual action control. As a mark of respect, the action control circuit "MrGO" is named after her — the letters M and R are the initials of Marly Roncken.

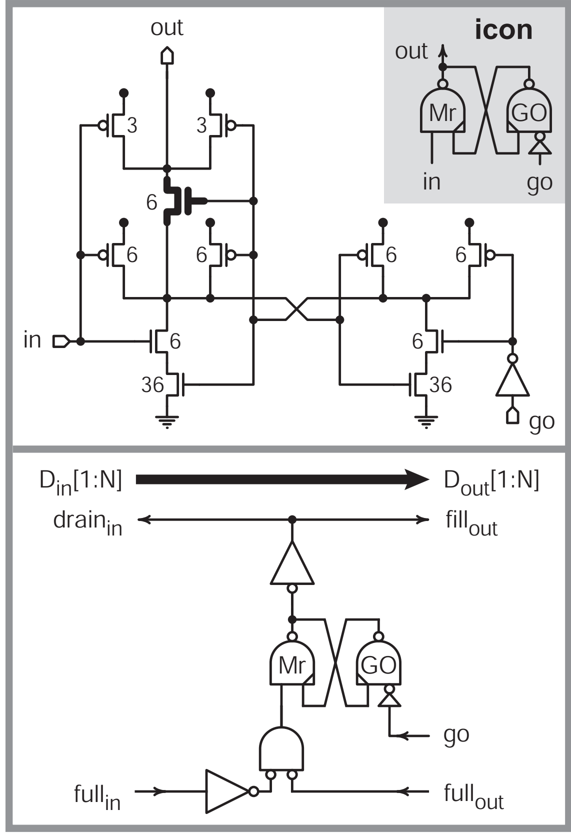


Fig. 5.2.1: MrGO with its icon inset in the grey area (top), and a joint with MrGO (bottom). The bold central transistor in MrGO delays the active-low grant signal, *out*, by conducting only after metastability ends. Transistor sizing reduces the logical effort from *in* to *out*. Split pull-up transistors 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 [44], proper stopper [37], and Seitz mutual exclusion element [20].

Selectively asserted *go* signals provide a wide variety of test options. The test options mentioned in sub-sections 5.3.1–5.3.3 and 5.3.7 are useful for structural testing, like the presence of stuck-at faults. The test options mentioned in sub-sections 5.3.3–5.3.6 are useful for testing delay faults and marginal functionality. For debug, all test options matter.

5.3.1 One-shot Test of a Selected Joint

Figure 5.3.1 gives a pictorial view of a one-shot test setup. 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. For example, the joint in Figure 5.2.1(bottom) will either do nothing or generate a high pulse on its fill and drain signals, changing both links. After re-freezing the selected joint, examination of its links and internal state reveals if it took the expected action. This is a typical test setup for structural testing of stuck-at faults.

5.3.2 Following a Thread of Actions

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. Figure 5.3.2 shows an example of a sequence of one-shot tests to cope with a marginal latch in one of the Anvil chips [46].

5.3.3 Breakpoints

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. Figure 5.3.3 shows a breakpoint test scenario taken from the Weaver chip [56, 46].

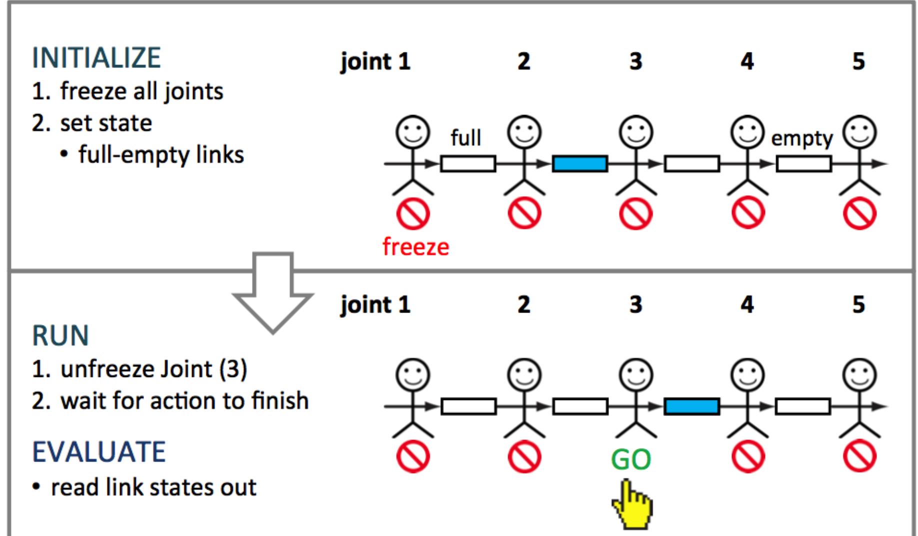


Fig. 5.3.1: Pictorial view of a one-shot test of selected joint 3. Full links are colored blue and empty links are colored white, as in Figure 4.1.1.

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]
⋮															
nogo	[1]	-	[2]	-	[3]	-	[4]	-	[5]	-	[6]	$\xrightarrow{F}$	[7]	-	[8]

Fig. 5.3.2: A series of one-shot tests shift the data items into place for the marginal latch test of Figure 5.3.7. 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 5.3.7. 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.

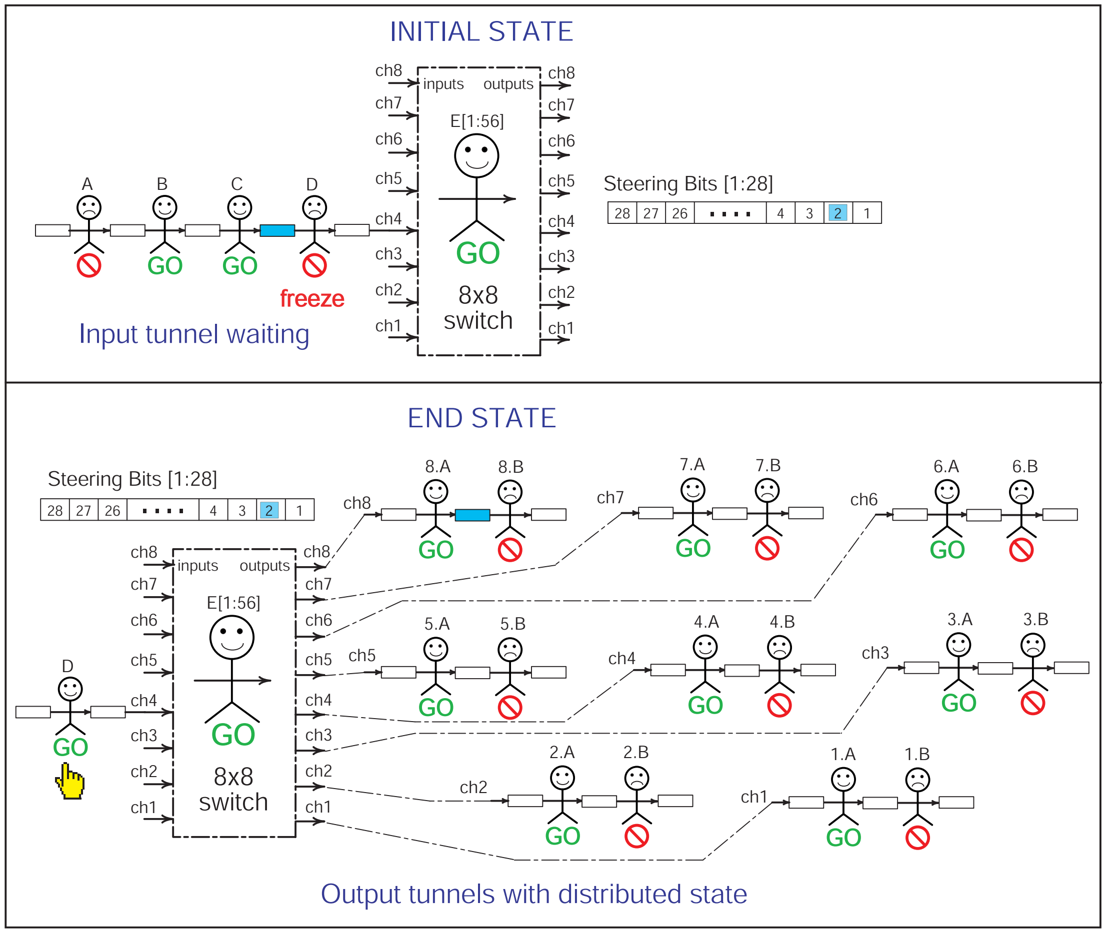


Fig. 5.3.3: Pictorial view of a breakpoint test in the Weaver chip to validate the crossbar's steering circuitry. As before, we color full links blue and empty links white.

5.3.4 Testing a Single Data Item at Speed

To test a single data item at speed, we leave 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. Figure 5.3.4 gives a pictorial view of such an at-speed test. Figure 5.3.5 gives a textural view of a similar at-speed test.

5.3.5 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 adequate data storage for the entire burst in the release and capture sections ahead of and behind the test section. Figure 5.3.6 gives a pictorial view of such a test. Figure 5.3.7 shows how we used such a test setup to identify a marginally operating latch in one of the Anvil chips.

5.3.6 Testing the Reverse Flow of “Bubbles” at Speed

Canopy graphs, like the one in Figure 4.3.1, teach us that data items flowing forward through a pipeline tend to move at a different speed than spaces or “bubbles” flowing backward, in the reverse direction. 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.

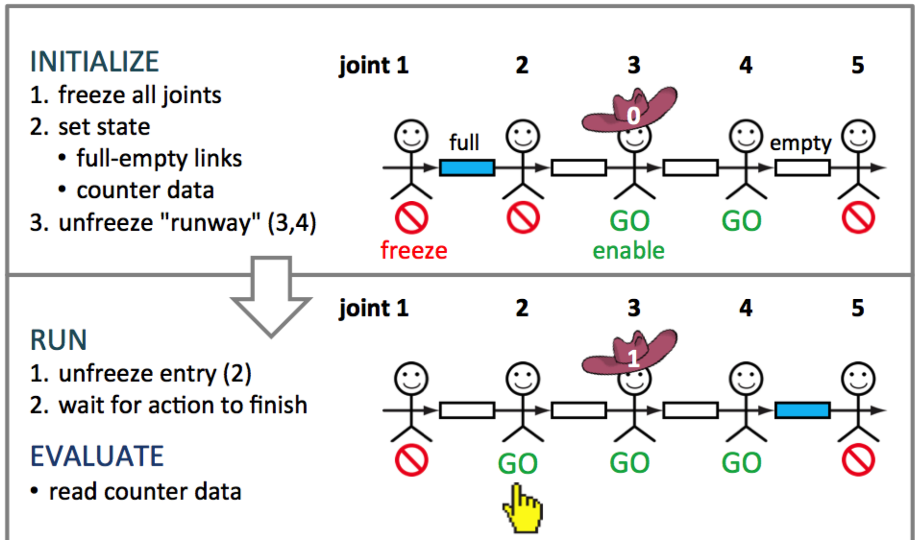


Fig. 5.3.4: Pictorial view of an at-speed test with a single data item. The data go at speed through joint 3, which has a counter that counts the number of actions performed by the joint. After the test, the counter value, which started at 0, should have a value of 1. As before, we color full links blue and empty links white.

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
⋮	[1]	-	2	-	3	-	4	$\xrightarrow{A}$	5	-	[6]	-	[7]	1
done	[1]	-	2	-	3	-	4	-	5	$\xrightarrow{A}$	[6]	-	[7]	1

Fig. 5.3.5: Textual representation of an at-speed test of a counter in joint 4 by sending a single data item through joint 4, 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, and all links between them are empty, illustrated by the simple dash "-" for each link. 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.

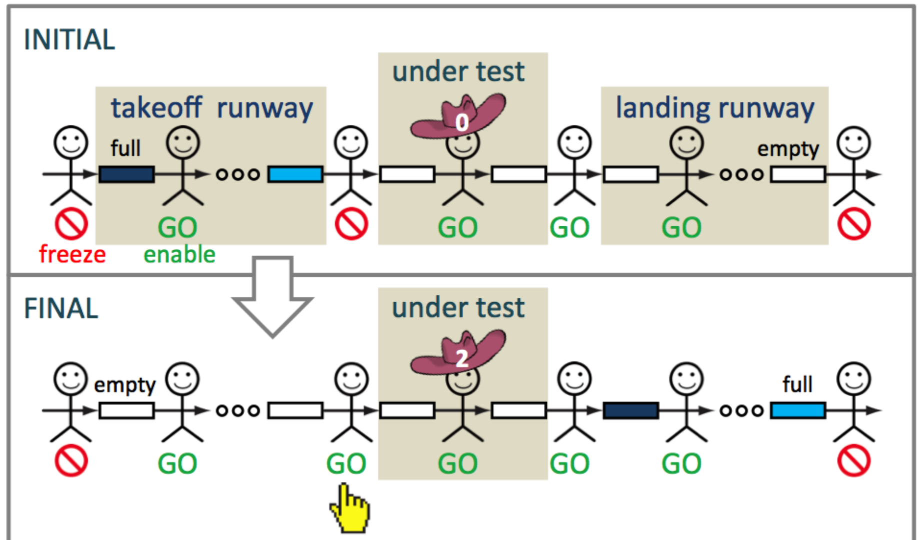


Fig. 5.3.6: Pictorial view of an at-speed test with a burst of data items. As before, we color full links blue and empty links white.

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]
⋮															
done	[1]	-	2	...	6	-	7	-	8	$\xrightarrow{A}$	9	...	13	$\xrightarrow{F}$	[14]

Fig. 5.3.7: At-speed test to identify a marginal latch near pipeline stage 8 using a burst of data items. Using the same notation as in Figure 5.3.5, 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 in one of its links — 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 5.3.2. 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.

A test section initialized with full rather than empty links reveals its response to bubbles, allowing detection of faulty behaviors often overlooked — see Figures 5.3.8 and 5.3.9.

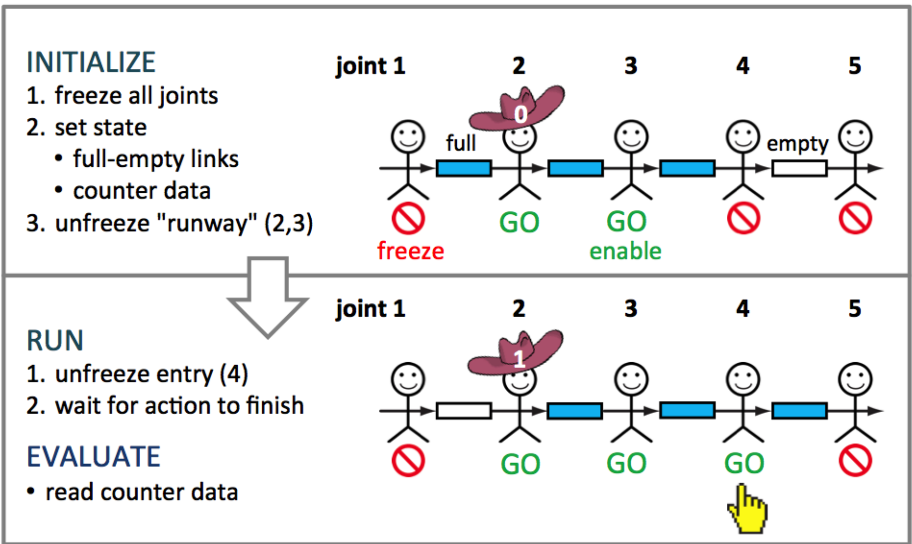


Fig. 5.3.8: Pictorial view of an at-speed test for bubbles. As before, we color full links blue and empty links white.

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
⋮	[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

Fig. 5.3.9: Textual representation of an at-speed test for bubbles. This test complements the test of data flowing forward in Figure 5.3.5 in that it tests the reverse flow. 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.

5.3.7 Stuck-at Fault Coverage

Figure 5.3.10 gives a pictorial view of the general setup for exhaustive stuck-at fault testing for links and joints. For each stuck-at fault, we (1) first freeze the joint to be tested, (2) then initialize the states in the links and the data, and (3) then permit the joint under test to go, allowing it to take at most one action. After re-freezing the selected joint, examination of its links and internal state reveals whether or not it took the expected action. We repeat this test for all relevant data and full-empty link combinations.

We do this for every (input-links)–joint–(output-links) configuration in the design.

A similar stuck-at test scheme can be defined for links with normally transparent latches.

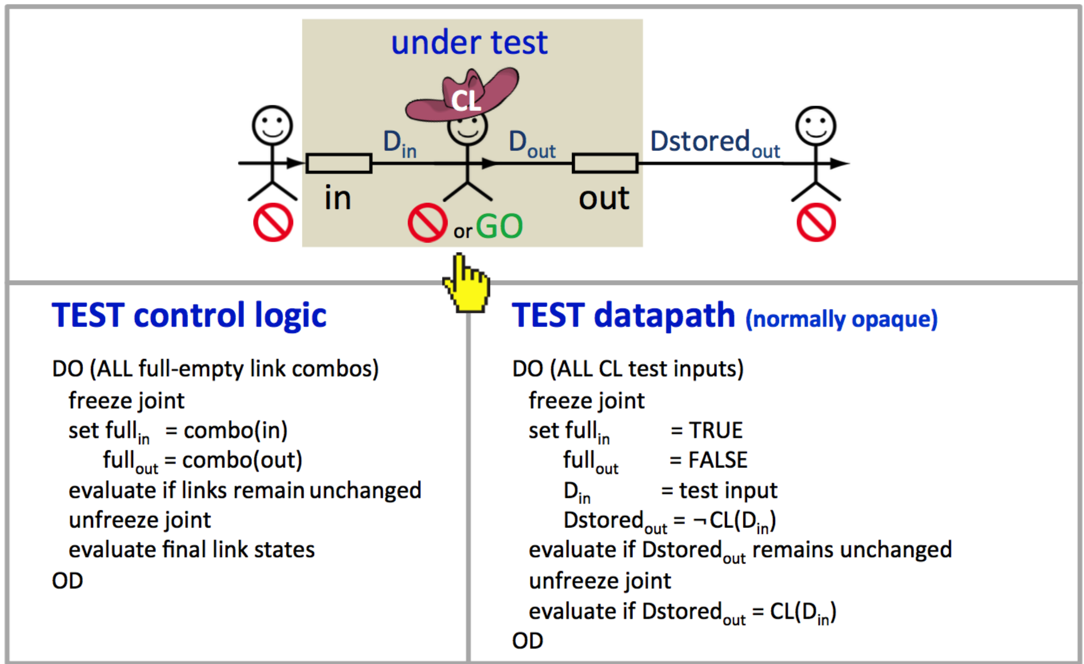


Fig. 5.3.10: Pictorial representation and test outline for exhaustive testing of stuck-at faults. As before, we color full links blue and empty links white.

5.4 Summary and Conclusion

The work outlined in this chapter presents the ARC's new point of view on testing. We emphasize that testing requires access to both *state* and *action*. 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, combined with traditional scan access to state, 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. The poster in Figure 5.5.1 illustrates this MrGO-centric test solution.

The next chapter implements this test solution and makes it ready for silicon test and debug, using traditional scan access and an industry-standard test interface.

5.5 My Contributions for Chapter 5

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

- I implemented MrGO, sizing its transistors using logical effort calculations for a typical FIFO joint configuration. I present a more precise sizing strategy in Chapter 7.
- I developed and ran test, debug and characterization scenarios for the ARC's Weaver and Anvil chips, using MrGO and the standard scan test interface presented in Chapter 6.

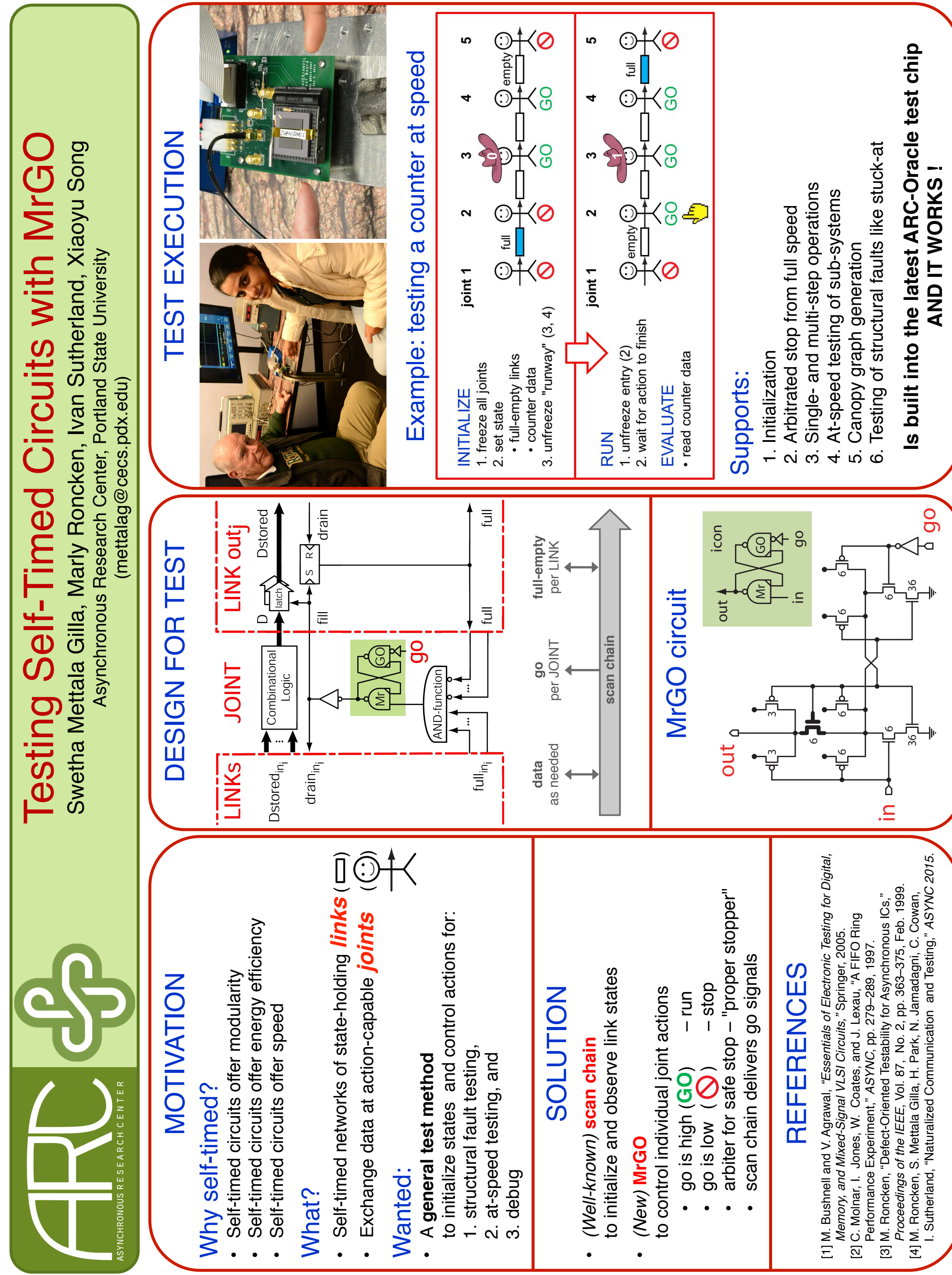


Fig. 5.5.1: My ASYNC 2015 poster illustrating how to test self-timed circuits using MrGO.

6

Hierarchical Design for Test and Debug

In this chapter, I give implementation details for the test and debug approach presented in Chapter 5. These implementations show how we can control the individual actions in a distributed self-timed system and how we can control and observe the distributed states. Specifically, Chapter 6 provides details on how we organize the scan actions, which we clock globally, to test an un-clocked self-timed system. We opted for a clocked scan solution, rather than a self-timed version, because a clocked version makes it easier to use existing standard scan test interfaces.

To stay in tune with the distributed operation and power distribution of our self-timed systems, we limit the global activity that a global scan clock may induce into the system. We limit global test activity in the system by scanning test stimuli in and test responses out through separate scan storage elements that play no role in the self-timed operations. If needed, surplus global test activity resulting from a globally scan-clocked write action of scan values into system signals can be partitioned over multiple scan-clock write actions. Likewise, surplus global test activity in the reverse direction resulting from a globally scan-clocked read action can be partitioned over multiple scan-clock read actions. Additionally,

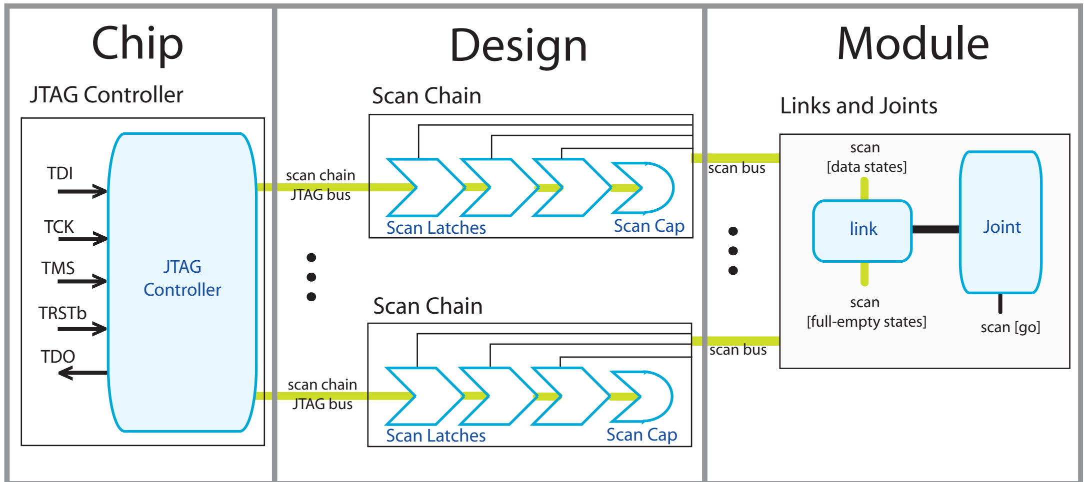


Fig. 6.0.1: Design for test hierarchy for self-timed systems, showing:
At the module level — right: A link with full or empty and data states, and a joint with *go* signals to start and stop individual joint actions.
At the design level — middle: Multiple scan chains of serially connected scan latches, each controlling or observing a state or *go* signal at the lower module level.
At chip level — left: The JTAG Controller to translate IO signals into scan chain instructions for (1) shifting test stimuli through IO signals into the scan chain, (2) writing these into state or *go* signals, (3) reading test responses from any resulting system actions out into the scan chain, and (4) shifting these out through the IO signals.

by blocking individual actions, we can prevent state changes from propagating further into the system until the desired test configuration is ready for execution. As such we are able to support standard clocked test interfaces and scan chains without jeopardizing the performance characteristics of the self-timed system.

Many commercial chips have a standard clocked test interface that can be programmed to control the scan operations as needed. For my thesis, I have taken a scan reference design from Sun-Oracle Laboratories as a starting point. This scan reference design consists of multiple clocked scan chains that are controlled through a standard clocked test interface as defined by the Joint Test Action Group (JTAG) [13].

The resulting design for test and debug approach is hierarchical. The lowest

level gives direct controllability and observability of state settings and state transitions, or actions, as explained in Chapter 5. The higher levels make the lower level controllability and observability settings accessible via the scan chains and the external JTAG test interface. The JTAG test interface requires only a small fixed number of test access pins on the chip. Figure 6.0.1 shows a block diagram of this hierarchy, which can be explained as follows:

1. **At the module level** I add and identify special control and observe signals in each link and joint to (1) initialize, or write, internal states, (2) start, single step, and stop actions, and (3) capture, or read, internal states.

2. **At the design level** I add serial-shift, parallel-read, parallel-write scan chains and hook them up to write the control and read the observe signals at the module level. More specifically, I use serial-shift scan chains that run independently and in parallel to the circuit under test. Each scan chain segment is composed of two latches per bit.

   Note that I separate the scan chain from the self-timed design that is under test. The standard scan test practice is to turn system latches or flipflops used in the design into scannable flipflops, where each flipflop will have two inputs. One input is devoted to the function need of the circuit and the other input is devoted to shifting data through the scan chain which includes the present flipflop. I avoid this standard scan test practice in order to maintain the state of the self-timed system during scan-shift operations, and to obtain a low-power scan profile.

   I connect the scan chains to the special control and observe signals at the module level. By doing so, I lose the direct controllability and observability of these signals, which now come under the controllability and observability capabilities of the scan chain operations. Though this costs test time, it favor-

ably reduces the number of external signals needed to control and observe the circuit states and actions.

3. **At the chip level**, I use a JTAG Controller configuration to program the scan chain operations. JTAG Controllers offer an external interface and configuration setup that are commonly used by chip manufacturers. Though this makes for an even more indirect controllability and observability of circuit state and actions, it provides a standard test interface through a small set of IO pins.

In the next three subsections I will give more detail on these three levels of test and debug hierarchy, and indicate how to avoid losing controllability or observability when moving from a direct non-standard test interface at the module level to an indirect standard interface at the chip level. Further details, including SPICE simulation waveforms, can be found in Appendices E–F. The GasP circuit examples in these appendices don't yet use links and joints, but the scan test solutions and results apply equally to link and joint circuit formulations.

The module-level solutions in this chapter are based on GasP circuits. Similar solutions can be constructed for Click circuits and Telescope GasP circuit.

6.1 Module level Test and Debug

Test and debug require the ability to control and observe the circuit state and to start, single-step or multi-step, and stop state-to-state transitions. The state of a GasP circuit is typically captured in (1) the full or empty statewires in its links and (2) the datapath latches. So, obviously, we'd like to control and observe statewires and datapath latches.

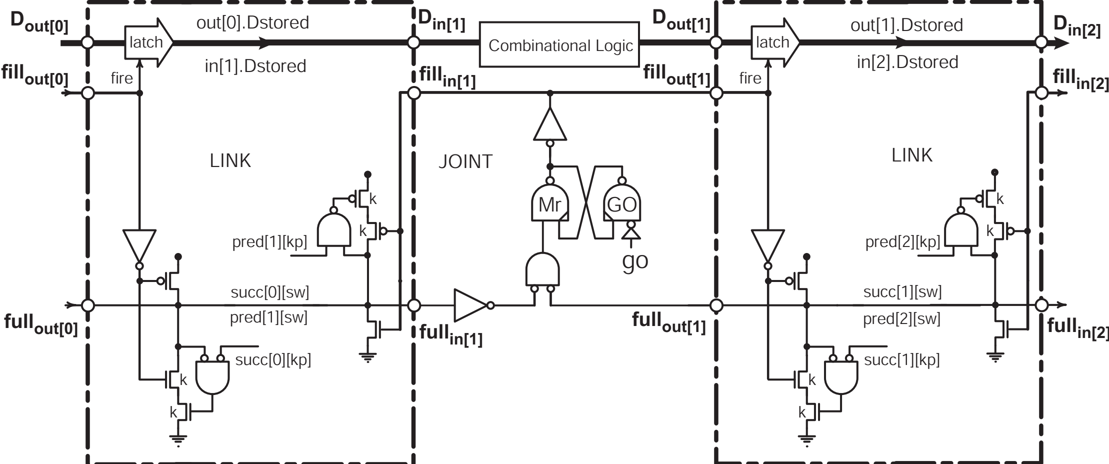


Fig. 6.1.1: First In First Out (FIFO) design example showing a joint with two GasP links. The links store data using 1-bit datapath latches. We would like to control and observe the following signals for initialization, test, and debug: the joint's *go* signal, each link's statewire and — if desired — its 1-bit stored data signal. To limit the size of the statewire drivers, we would like to turn off the keepers while we drive the statewire during test operations. The PMOS or NMOS type AND gates at the link's high and low keepers and their *kp*-labeled input signals allow us to do that — see also Figure 6.2.3.

From a 1997 publication by Charles Molnar et al. [37] and a 2010 ARC report by Ivan Sutherland [52], we know that to debug GasP we must stop a self-timed operation in full flight, which requires a so-called *Proper Stopper*. From the ARC's 2015 publication by Marly Roncken et al. [46] and Chapter 5, we know to add a *Proper Starter* as well, so that we can force the circuit to be stable when we start a normal or test operation. MrGO of Figure 5.2.1 from Chapter 5 combines both features in one proper start-stop design.

As an example, Figure 6.1.1 shows a GasP First In First Out (FIFO) design with a joint and two links, and a summary of the signals I'd like to control and observe for initialization, test, and debug. MrGO is inserted in the joint, after the synchronizing PMOS AND-type gate and at or before the gate that generates the fire signal. MrGO receives an arbitrated start-stop signal, called *go* here. A

high go signal allows the GasP module to operate and a low *go* signal stops the self-timed action in a clean way.

To coordinate the initialization of state signals, full-empty and data, with the setting and resetting of proper start and stop signals, it is necessary to use an initialization-start-stop protocol that is in-sensitive to delays in *go* signals. Intuitively, the protocol separates low from high test settings for *go* signals, by requiring that:

- All *go* signals return to low at the end of each test, before we scan in new full-empty and data state variables for the next test.
- When the new scanned-in state of the circuit has stabilized, a subset of the *go* signals may be set to high, monotonically, either in parallel or serially.

The exact formalization and mathematical foundation of this initialization-start-stop protocol needs further work and is left for future research.

Having separate state initialization and action start signals is new in the context of GasP circuits and in the context of test and debug. But it's not a new idea in itself, in the context of design: the self-timed circuits generated by the Philips Tangram compiler in the early 1990's already separated initialization from start [60]. Surprisingly, though, people seem to have forgotten the need for this. During the recent ASYNC conference, held in Potsdam from 12-14 May 2014, one presenter addressed the issue of being unable to initialize, specifically reset, his self-timed circuits [17]. His mistake was that he used a single signal to reset and to start the circuit. As a result, his circuits lacked the necessary stabilization time window between exiting the reset phase and starting the circuit. Such a stabilization time window is necessary because exiting an initialization phase, reset included, takes time. For instance, it takes time to go from driving a GasP

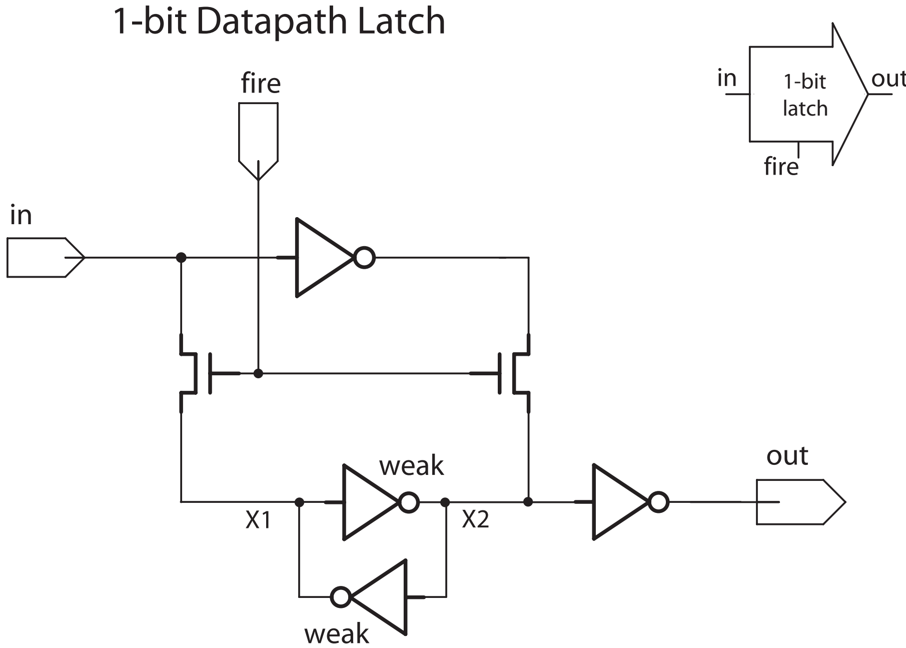


Fig. 6.1.2: Implementation of a 1-bit datapath latch. Its icon is shown in the top right corner. Where needed, we replace the 1-bit latch with the 1-bit scan latch of Figure 6.2.4(top).

statewire externally to handing over the drive responsibility to the statewire keepers and to disconnect the external drive from the statewire. Moreover, until the external drive is properly disconnected, the external driver can interfere with an already running circuit operation that's driving the statewire. By providing a separate start signal, the start of the circuit operation can be postponed until the circuit is stable and properly disconnected from its reset and initialization drivers.

In addition to what's shown in Figure 6.1.1, a joint may have extra data signals, coming from data latches in the links, which act as extra control inputs to the joint's gates. There may also be extra control signals going the other direction, which act as selection signals for the datapath. We can manage these by controlling and observing the full-empty state and data latches in the links.

Figure 6.1.2 shows the circuit of a 1-bit data latch.

The subsections below give a more detailed summary of the control and observation signals in Figure 6.1.1, and how they help in test and debug.

6.1.1 Proper Start Control: Make *go* High

Each GasP joint has its own *go* signal to permit the next action and the corresponding state transitions in the links from the present link states, as explained in Chapter 5. By making the *go* signal high, we allow the joint to act and change the state of its links, where applicable. The GasP joint in Figure 6.1.1 can act if its predecessor link, *in[1]*, is full ($full_{in[1]}$ high) and its successor link, *out[1]*, is empty ($full_{out[1]}$ low). The AND of *go* high, $full_{in[1]}$ high, and $full_{out[1]}$ low results in $drain_{in[1]}$ and $fill_{out[1]}$ going high, which again results in three concurrent sub-actions that together copy the data from *in[1]* to *out[1]*:

1. $fill_{out[1]}$ high makes *fire* high, copying $D_{out[1]}$ to *out[1]*'s data latch and *out[1].Dstored*,
2. $drain_{in[1]}$ high makes $full_{in[1]}$ low, declaring link *in[1]* empty, and
3. $fill_{out[1]}$ high makes $full_{out[1]}$ high, declaring link *out[1]* full.

These three actions are followed by a 5-gate-delay self-resetting phase, starting with $full_{in[1]}$ low and $full_{out[1]}$ high, resetting the AND gate to low, and ending with the following three concurrent actions:

1. fire goes low, making the latch in link *out[1]* opaque, a.k.a. "closing the latch," which renders the latch insensitive to new data changes on *in[1].Dstored*,

2. $drain_{in[1]}$ goes low, thus enabling the other side to fill link *in[1]* again with new data,

3. and $fill_{out[1]}$ goes low, thus enabling the other side to drain link *out[1]*.

Note that the high *go* signal does not interfere with the self-reset phase. This is crucial because it keeps intact the existing handshake protocol on the statewires. As a result, it is possible to test not only a single-step state transition as described above, but arbitrarily long sequences of state transitions in a GasP pipeline. This is extremely useful for debug and characterization of throughput and power.

6.1.2 Proper Stop Control: Make *go* Low

To stop a self-timed operation cleanly, we arbitrate between continuing the joint operation and stopping it. We stop the operation by making *go* low at some point in time. This can be when the circuit is stable or while it is running at full speed.

The arbitration circuit in MrGO is fair in the sense that a signal that loses arbitration will be granted during the next arbitration cycle. So, if the low *go* loses the arbitration because MrGO's arbiter favored continuing the self-timed operation, it will win the next arbitration cycle. Moreover, *go* low will win the arbiter within one non-blocking action cycle, here: one arbitration delay and 10 gate delays later.

As an example, consider a pipeline with eleven GasP links connecting eleven GasP joints, like those in Figure 6.1.1, eight of which have permission to act and three of which are blocked:

```
A|B.C|-.-.-.-.-|-.-.-.
```

where

- The letters A, B, and C denote the data in full links.
- A dash "−" denotes an empty link.
- A period "." represents a GasP joint with permission to act (*go* is high).
- A vertical bar "|" represents a GasP joint not permitted to act (*go* is low).

The values in A, B, and C are waiting to enter the long center section of this pipeline. If we unblock the second blocked GasP joint we get:

```
A|B.C.-.-.-.-.-|-.-.-.
```

which is unstable and rapidly changes to the stable configuration:

```
A|-.-.-.-.-.B.C|-.-.-.
```

What's interesting about this example is that B and C move at full speed from one link to another. Separate control of the *go* signal to each joint permits us to test short bursts of activity at full speed. See also Figure 5.3.6 in Chapter 5.

ARC report ARC2013-smg09 in Appendix F presents a similar but slightly smaller example that controls the *go* signals from the higher level JTAG controller and scan chains. The appendix shows SPICE level waveforms for the rapidly-changing statewires and fire signals between the initial and final pipeline configuration.

6.1.3 Full-empty State Control and Observation

We can initialize links to either full, which indicates the presence of valid data, or empty, which indicates the absence of data or that the data are no longer needed

and may be changed by the sender. For instance, initializing *pred[1][sw]* to high for the GasP link in Figure 6.1.1 makes link *in[1]* full, thus indicates that there's new data on the in[1].Dstored signal stored in the 1-bit latch in link *in[1]*.

In addition to controlling the state, I'd also like to *observe* the state of each statewire – without destroying it. Being able to single-step through a computation and inspect the state at each step on the way, without destroying it, will be particularly helpful for debugging.

6.1.4 Data Control and Observation

Though we may not want to control or observe all datapath latches, we might want to control and observe at least the ones that interface with the control logic, i.e.,:

- control datapath latches whose outputs control a joint action
- control signals, other than fire signals, that steer datapath operations, e.g., control signals for data multiplexers.

Also here, I'd also like to observe the state signals without destroying them, to accommodate debugging efforts.

6.2 Design Level Test and debug

At this level, we'll add extra connections to the individual *go* signals of each joint and to the individual full-empty statewire signals and selected data signals of each link, so they remain easy to access but can be controlled and observed through a much smaller external interface. As access mechanism, I will use

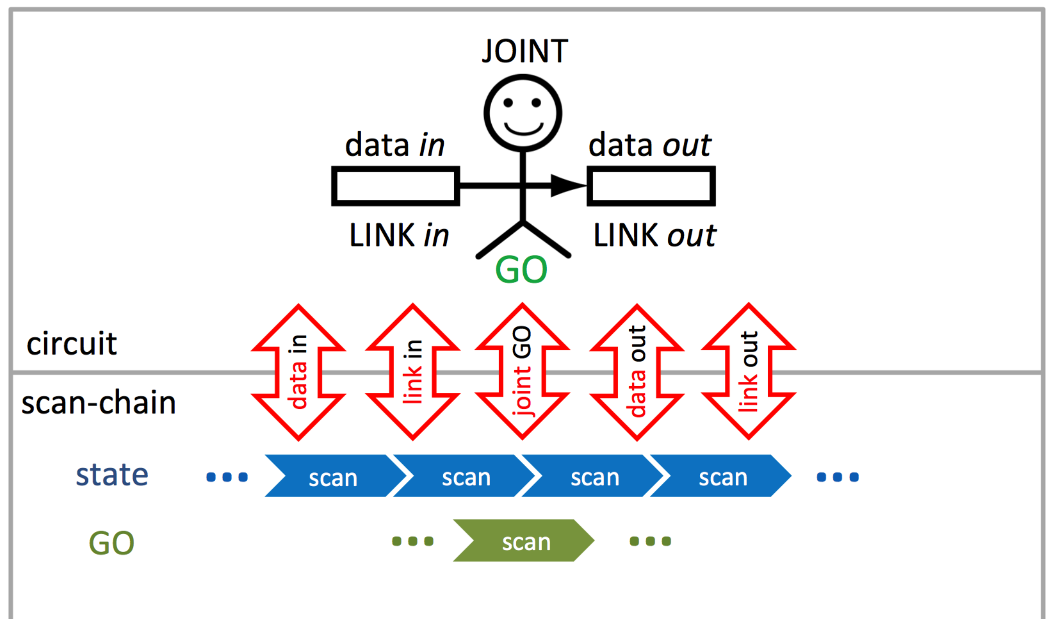


Fig. 6.2.1: Pictorial representation of a GasP FIFO design (top) with two scan chains (bottom). The stick figure at the top represents the FIFO joint and the rectangular boxes represent the FIFO links. Each scan chain consists of serially connected scan segments. Each scan segment connects to exactly one joint, link or data signal. The one-to-one connections are indicated in the bidirectional vertical arrows between circuit and scan chains. The upper scan chain connects to state signals in the circuit, including those in the pictured links. The lower scan chain connects to *go* signals in the circuit, including those in the pictured joint. As in Figure 4.1.1, we color the rectangles of full links blue, and those of empty links white.

serial shift and parallel read-write scan chains that run along the joints and links and that connect to the control and observation signals for *go*, full-empty, and data, discussed in Section 6.1.

Figure 6.2.1 gives a pictorial representation of a self-timed GasP FIFO connected to two such scan chains. The upper scan chain is for controlling and observing state signals in the circuit, including state signals of this FIFO circuit. The lower scan chain is for controlling — and possibly observing — *go* signals in the circuit, including the *go* signal of this FIFO circuit. The stick figure at the top represents the FIFO joint and the rectangular boxes left and right to the stick

figure represent the FIFO links and their data. The series of scan elements at the bottom form the scan chains. The vertical bidirectional arrows indicate the read-write connections between circuit and scan chains.

Figure 6.2.2 shows how these connections can be used to start and stop circuit actions and to write and read the state values that we want to control and observe. The series of scan operations in Figure 6.2.2 illustrate how each scan chain can:

- **shift bit values serially in (load) and out (unload):**
  In the design hierarchy in Figure 6.0.1, the scan chains are the interfaces between the JTAG bus connections and the scan bus connections. In Figures 6.2.1–6.2.2, the JTAG bus connections are represented as scan inputs and outputs, and the scan bus connections are represented as the vertical bidirectional arrows between circuit and scan chains. Each vertical arrow is a one-to-one connection between a circuit signal and a scan segment. The scan segments do the serial shifting.

- **read, or capture, bit values in parallel from the design into the scan chain:**
  In Figures 6.2.1–6.2.2, the values to be read travel down the vertical arrows from the target circuit signals to the corresponding scan segments.

- **write, or launch, bit values in parallel from the scan chain into the design:**
  In Figures 6.2.1–6.2.2, the values to be written travel up the vertical arrows from the corresponding scan segments to the target circuit signals.

The scan operations in Figure 6.2.2 support a one-shot test of the FIFO for copying the data forward from its data input link *in* to its data output link *out*,

as illustrated in Figure 4.1.1 of Chapter 4. Figure 5.3.1 of Chapter 5 showed the principles behind such a one-shot test, but without the corresponding scan operations for enabling and disabling the *go* signal and for reading and writing the state signals.

The narrative for Figure 6.2.2 is as follows. First, we stop all FIFO actions by making the *go* signal in the FIFO's joint low. To do this, we shift the value low (0) into the appropriate scan segment (steps 1 and 2) and then write (load) low into *go* (step 3). With all circuit actions stopped, it is now safe to change the circuit states. In step 3, we shift into position the values *1*, *full*, *0*, and *empty* for the full-empty GasP statewire and data signals of respectively link *in* and link *out*, In step 4, we write these values into the target signals. We are now ready to test if the FIFO will forward the data from its full link *in* to its empty link *out*.

To enable the FIFO to act, we must first make its *go* signal high. In steps 6 and 7, we shift the value high (1) into the appropriate scan segment and then write it into the *go* signal. If the circuit acts correctly, it will copy the data from *in* to *out*, drain *in*, and fill *out*, resulting in step 8 — see also Figures 4.1.1 and Figure 5.3.1. To validate that this action happened, we must scan out the full-empty statewire and data values. To do this, we first stop all FIFO actions (steps 9 and 10), then read the FIFO state values of both links (step 11), and shift them out for inspection (step 12).

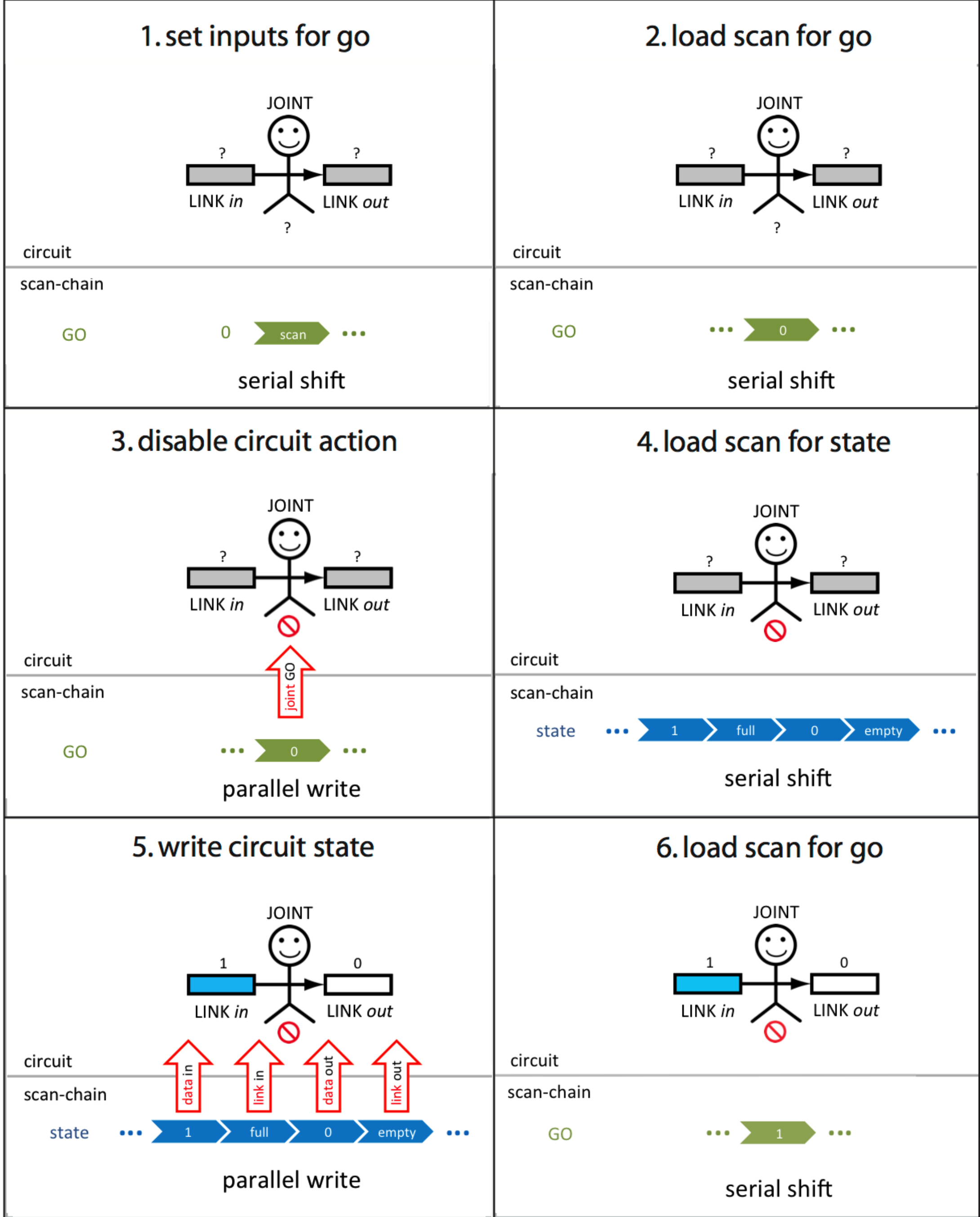


Fig. 6.2.2: Scan operations for a one-shot FIFO test — see also Figure 5.3.1.

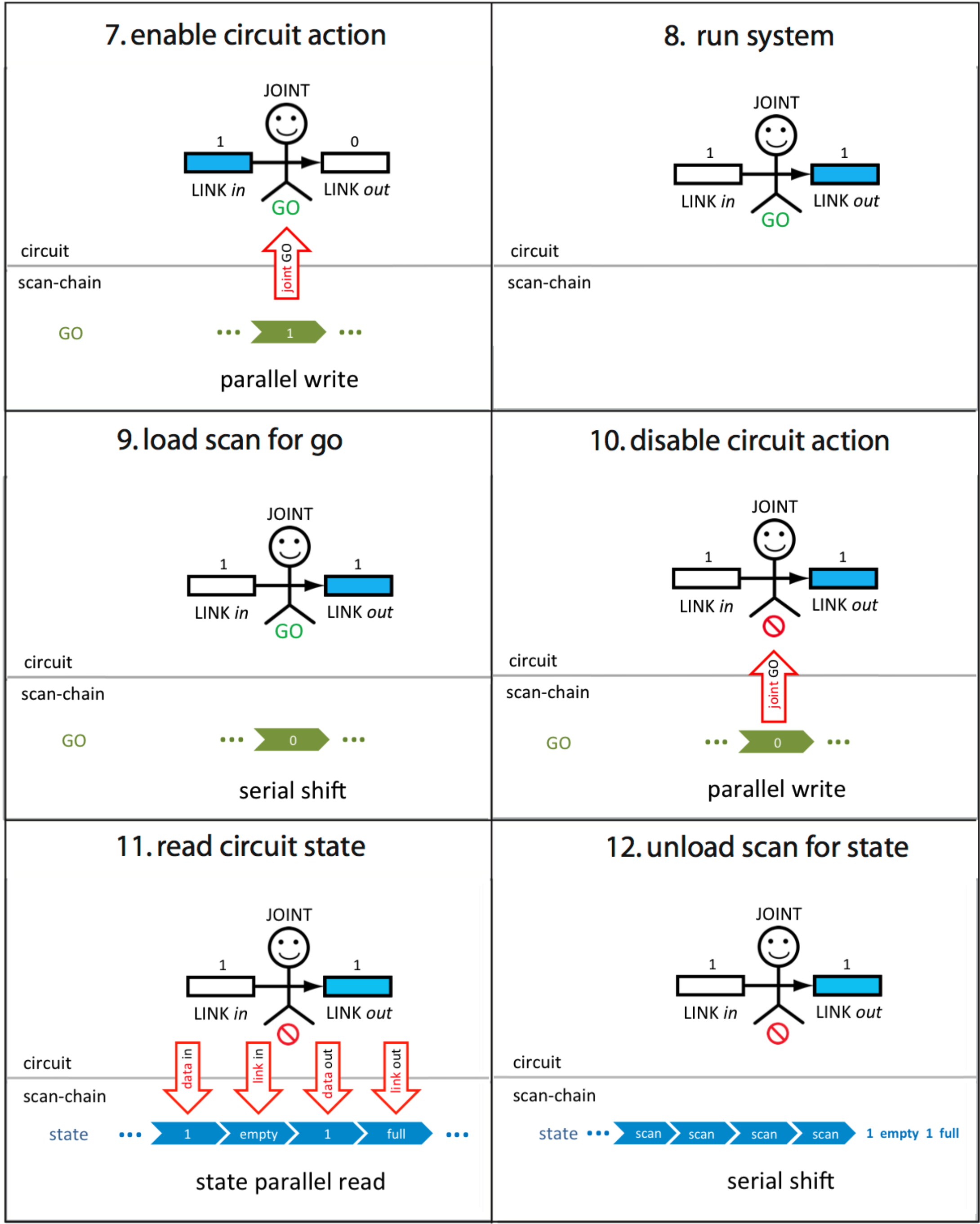


Fig. 6.2.2: (continued)

Figure 6.2.2 shows clearly that the serial shifting takes place inside the scan chain but outside the circuit. The only interaction between the circuit and the scan chain is during parallel read or write operations.

Note that I use the scan chain to disable the FIFO's *go* signal, making *go* low. I prevent the FIFO from acting **before** I allow the scan chain to read from or write into any of the FIFO's state signals. Likewise, I use the scan chain to enable the FIFO's *go* signal, making *go* high. I enable the FIFO to act **after** the FIFO state is updated and stable.

To be able to stop and start FIFO actions separately from initializing FIFO states, the scan chain for the *go* signals is separate from the scan chain for the state signals. This is the reason for the two scan chains in Figures 6.2.1–6.2.2. Alternatively, I could have used a single scan chain with extra scan segments, say one per already existing *go* or state scan segment. I would use the extra scan segment to indicate whether or not read or write actions should ignore and thus disconnect the associated *go* or state scan segment from the circuit. Another alternative, used in the Weaver, is to have one scan chain but two separate write signals, one for *go* signals and the other one for the state signals.

The scan segments that we use for testing GasP circuits come in three flavors, shown in Figures 6.2.3–6.2.4. The first flavor connects to signals that are *never* driven by the self-timed design, such as the *go* signals in the joints in Figures 6.2.1–6.2.2. Each such scan segment drives and keeps the scan-launched value on the target signal. The *drive-and-keep latch* in Figure 6.2.3(top) implements such a scan segment. The second and third flavors connect to signals that are under control of the self-timed design, such as the *full* and data signals in a link. The second flavor of scan segments is for signals like the full-empty statewire in a GasP link, where we want the driver circuitry of the signal to op-

erate in mutual exclusion with its — possibly strong — keeper circuitry. The *drive-and-release latch* in Figure 6.2.3(bottom) implements this second flavor scan segment. The third flavor is for signals, like a stored data signal in a GasP link, where we allow the driver circuitry to overtake the value on the — necessarily weaker — keeper. The *shift-and-capture latch* in Figure 6.2.4(bottom) implements this third flavor scan segment.

The fat green lines in the three designs are the scan buses. They bundle the relevant scan signals communicated from one scan segment to the next into a uniform communication link. Each bus contains the scan clocks for shifting and reading and writing, and the scan data bit that's shifted forward, as well as the final scan data and clock signals coming back.

Further details about the implementations and notations used in Figures 6.2.3–6.2.4 follow in Sections 6.2.1–6.2.4 below and in Appendix E.

6.2.1 Scan Shift

To observe GasP signals without destroying them, the serial scan shifts are done outside the GasP circuit, using an LSSD type scan shift register consisting of two latches in series clocked by non-overlapping clocks [6].

Each latch pair "small W" and "small R" in Figures 6.2.3–6.2.4 acts as a master-slave shift register. The latches are clocked by two active-high scan clocks, *phi1* and *phi2*. When its scan clock is high, the value at the latch input is copied into the latch and onto the latch output. When the scan clock is low, the latch keeps its currently stored value.

The last scan chain segment in a single scan chain is connected to a so-called

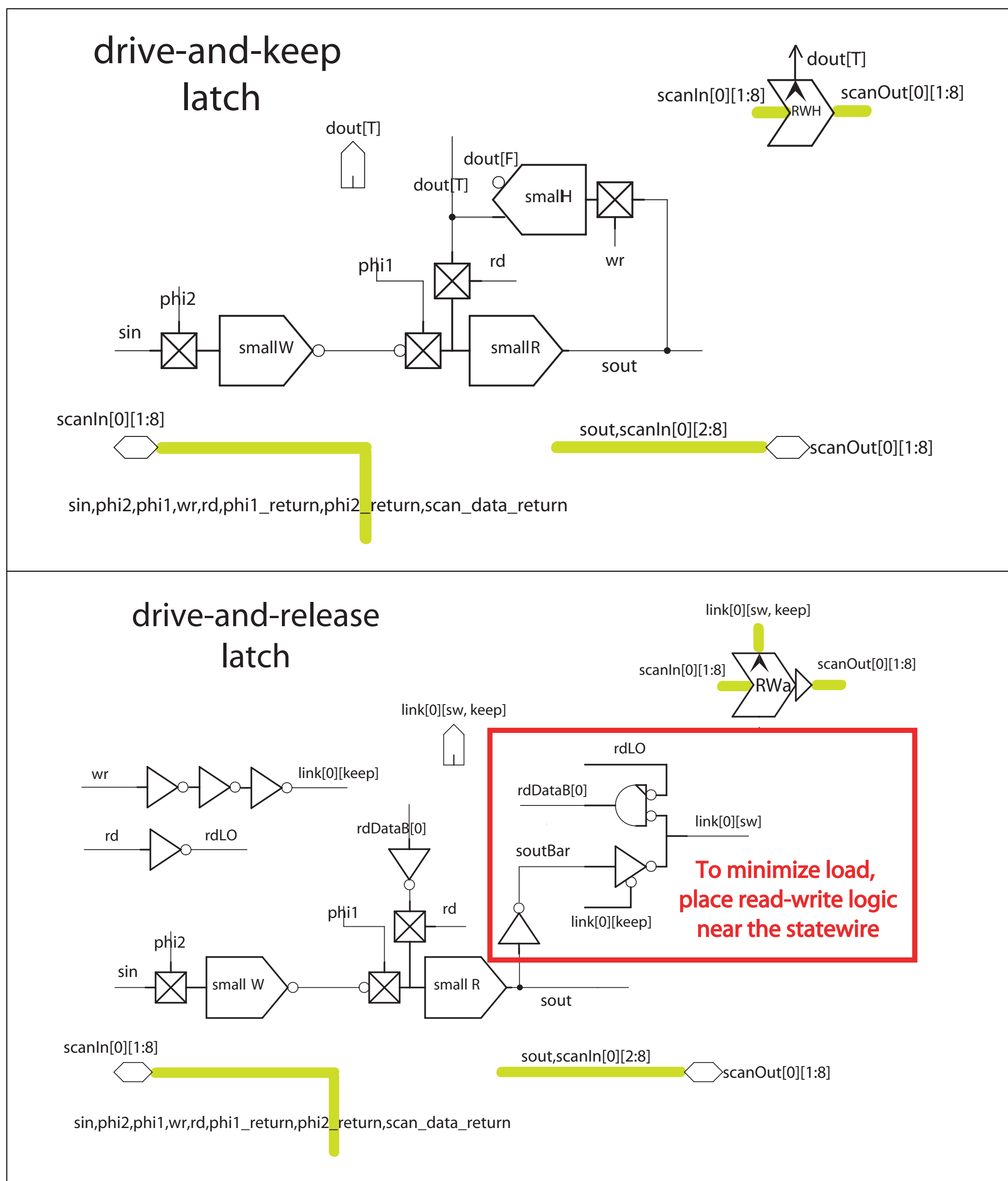


Fig. 6.2.3: First (top) and second (bottom) flavor scan segment implementations for GasP, for connecting *go* signals respectively full-empty statewire signals. To control or observe a *go* signal, e.g., the *go* signal of the joint in Figure 6.1.1, connect it to signal *dout[T]* of the drive-and-keep latch. To control or observe a full-empty statewire driver-keeper pair, e.g., the driver and keeper signals *succ[1][sw]* and *succ[1][kp]* in Figure 6.1.1, connect the pair to the pair of signals *link[0][sw,keep]* of the drive-and-release latch. The scan segments can be connected in series to form one or more scan chains. Scan values are shifted in from *scanIn[0][1:8]* to *scanOut[0][1:8]*, with *scanIn[0]* (*sin*) representing the scan input value.

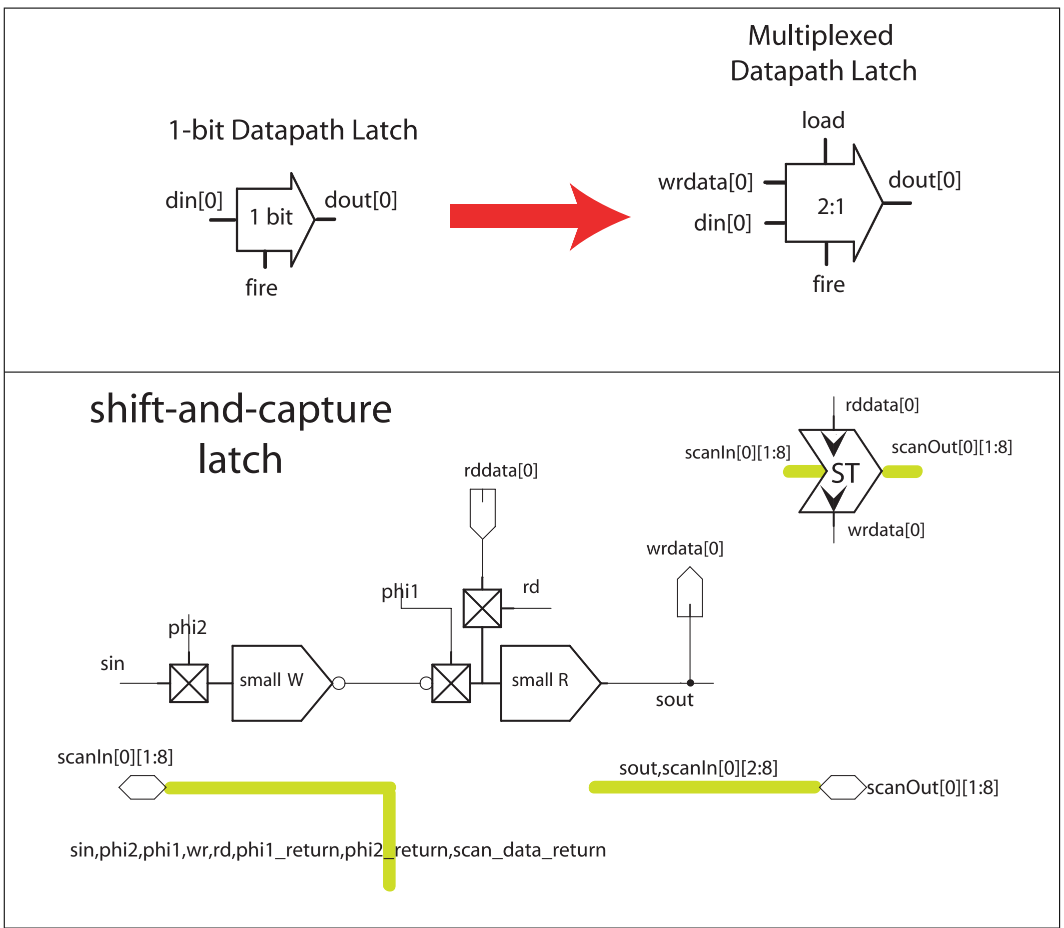


Fig. 6.2.4: Single-bit datapath latch with its multiplexed datapath latch version (top) and a corresponding third flavor scan segment implementation (bottom). The top-left datapath latch cannot be controlled but it can be observed through scan by connecting its *dout[0]* signal to *rddata[0]* of the shift-and-capture latch. The top-right multiplexed datapath latch can be controlled and observed through scan by connecting its signals *wrdata[0]*, load, and *dout[0]* to the respective signals *wrdata[0]*, *wr* (*scanIn[0][4]*), and *rddata[0]* of the shift-and-capture latch. By connecting multiple scan segments in series we can form one or more scan chains. Scan values are shifted in from *scanIn[0][1:8]* to *scanOut[0][1:8]*, with *scanIn[0]* (*sin*) representing the scan input value.

ScanCap design (see Figure 6.0.1) which returns three of the scan signals, *phi1* and *phi2* and *sout* respectively, and sends these back into the scan chain in reverse direction, as *phi1_return* and *phi2_return* and *scan_data_return*. These three signals travel back, un-clocked, to the first scan segment and are visible as scan output pins.

Signal *scan_data_return* is the *sout* signal of the entire scan chain. It is returned in reverse direction by the scan chain's end segment, ScanCap, and output under signal name *scan_data_return*. As *scan_data_return*, it can be inspected for test and debug.

Signals *phi1_return* and *phi2_return* are the signals at the end of the farthest out clock branches for scan clocks *phi1* and *phi2*. Each return signal acts as a completion detection signal that indicates that the clock signal has reached all end points in the distributed clock network. We use these return signals to generate non-overlapping-high *phi1* and *phi2* clocks for reliable scan shifting.

6.2.2 Non-Destructive Read of Circuit Signals

The slave part of the shift register, "small R" in Figures 6.2.3–6.2.4, can additionally copy and store the value of the GasP signal that it's connected to, so the value can be shifted out for inspection during test and debug. This copy action happens when the scan read clock signal, rd, is high, and has no impact on the value of the GasP circuit signal other than adding extra load capacitance. Because full-empty statewires may be long, we minimized the extra load capacitance on the statewire by splitting the read action over two separate circuits — see Figure 6.2.3(bottom): one near the statewire that inputs a local signal whose value matches that of the statewire ANDed with the read clock, and another one

to capture this ANDed signal into the slave latch of the shift register.

6.2.3 Non-Destructive Write of Circuit Signals

The scan segment designs differ slightly in how they write the GasP circuit signals. The designs in Figure 6.2.3(top) and Figure 6.2.4(bottom) copy the value of scan signal *sout* into the circuit signal when the scan write clock, *wr*, is high, and keep it there when *wr* is low. The design in Figure 6.2.3(bottom) copies *sout* into the statewire three gate delays after *wr* goes high, and uses these three gate delays to disable the statewire keepers. It releases the drive three gate delays after *wr* goes low, and uses these three gate delays to transfer the drive responsibility back to the statewire keepers. The three gate delay skew matches the delay in the enabling and disabling circuitry in the keeper logic, shown in Figure 6.1.1.

6.2.4 Initialization at Power-Up

Like the direct access configuration described in Section 6.1, the scan chain approach described here still allows us to read the state of the GasP circuit without destroying it, and still allows us to access all state variables. Control and observation are more indirect, though, because only the scan bus signals at the interface of the first scan segment in the scan chain are visible to the test environment. The advantage of this is a drastic reduction in external test pins. The disadvantage is that each control and observation action must be translated into the correct number and sequence of scan clock operations for scan shifting, reading GasP circuit signals, and writing circuit signals. As a result, some operations that we used to take for granted when we had direct

access to the GasP state signals require more planning.

The key operation that requires more planning is initialization at or shortly after power-up. This is especially desirable for GasP circuits, which may experience fights on the statewires when both the high and the low statewire drivers happen to get activated at power-up. In the past, each GasP module had a global master-clear signal that, when high, drove all statewires low, see for instance Figure 2.2.1 in Chapter 2. Having all statewires low yields a stable, well-defined initial state at or shortly after power-up. The scan chain can perform a similarly quick reset action if, for all scan chains, we make both scan shift clocks, *phi1* and *phi2*, high, and scan write clock, *wr*, high, and scan input signal, *sin*, low. This will cause the low *sin* signal to ripple through each scan chain to all *go* signals and statewires. We can do the same for scanned data latches, if needed.

6.3 Chip Level Test and debug

The external scan chain interface presented in Section 6.2. is still rather exotic compared to standard commercial scan interfaces. We can create a standard interface by connecting the scan chains to the IO pins via a standard scan interface that's used in industry and that can be programmed to control the scan operations as needed.

As standard scan interface, I use the JTAG Controller defined by the Joint Test Action Group (JTAG) [13]. The JTAG Controller is an IEEE 1149.1 standard for Standard Test Access Port (TAP) and Boundary Scan. Boundary scan test focused originally on controlling the IO pins of each chip for the purpose of testing the interconnections between chips on a board. But the test patterns stored at the chip boundaries can also be used to test the individual chips.

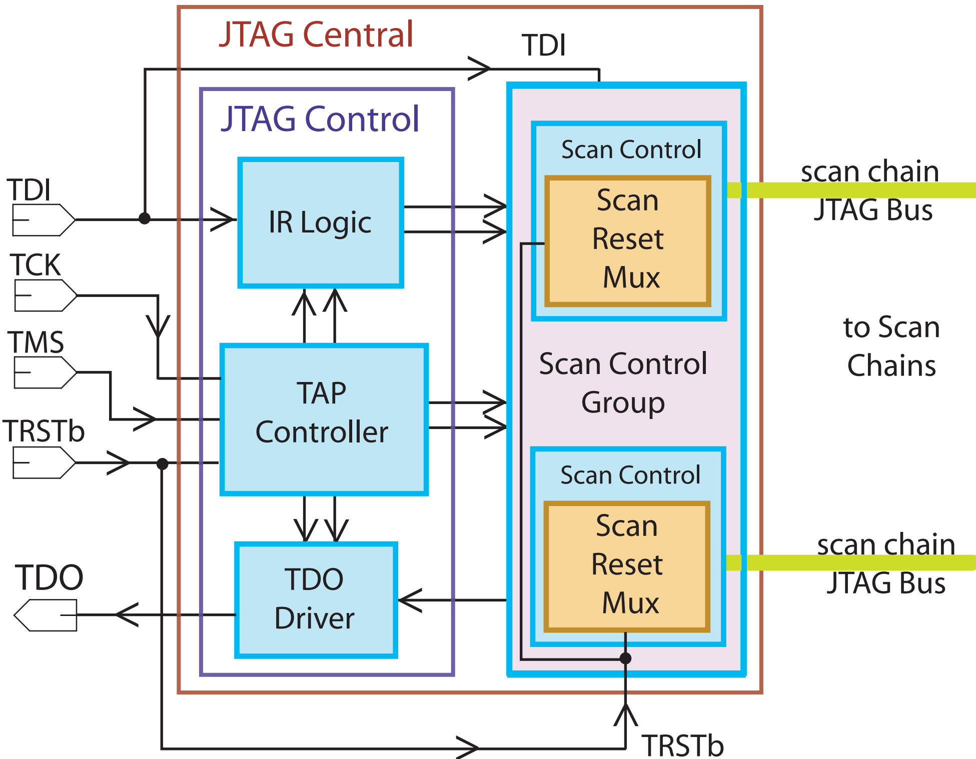


Fig. 6.3.1: Block diagram of the JTAG Controller, which observes the output results for the various scan chains via one test output, TDO (Test Data Out), and which controls the scan chain operations via four external test input pins: TDI (Test Data In), TCK (Test Clock), TMS (Test Mode Select), and *TRSTb* (Test Reset active low signal).

The JTAG test view fits with our self-timed circuit test view: we can view the design as the board, the joints as the chips, and the links as the interconnections between the chips on the board. Likewise, scan chains can be viewed as boundary scan chains, where the serial scan segments store the test patterns and where the write-read scan connections between scan segments and circuit launch the test stimuli and capture the test responses.

Figure 6.3.1 gives an overview of the architecture of the JTAG Controller, showing the following modules:

- JTAG Central: This is the interface module between the external pins and the GasP scan chains. It has two submodules: JTAG Control and Scan Control Group.
- JTAG Control: This module contains the IEEE 1149.1 standard Test Access Port (TAP) Controller, which is a finite state machine from which we can program the Instruction Register (IR), which keeps test related state information for the TAP Controller, the Test Data Output (TDO) Driver, which determines which test data goes to the TDO pin, and the Scan Control Group, which uses the stored IR and the TAP outputs to select a scan chain and apply IR and TAP instructions to it.
- Scan Control Group: This module groups the Test Data Input (TDI) signal with the TAP controller and IR logic signals coming from the JTAG Control module. It translates these into specific scan chain signal settings and hands a TDO signal back from the scan chains to the JTAG Control module.

I was lucky to have a reference design from Sun-Oracle Laboratories with a JTAG Controller implementation. I studied this design carefully and adapted it

so GasP circuits with links and joints can be initialized at power-up. The original Sun-Oracle solution used a global reset signal built into the design to initialize the circuit during or after powering up the circuit, as we did in our traditional (T)GasP circuits before we re-designed them with links and joints [46] — see for instance signal *in[mc]* in Figure 2.2.1 of Chapter 2.

Getting the links and joints in a stable state during power-up can be done by overlapping the high signals for the normally non-overlapping scan clocks, while keeping the write clocks high, and while keeping all scan inputs low for the scan chains connected to *go* and full-empty statewire signals. In addition to initialization at power-up, overlapping scan shift clocks can also speed up operations that are an integral part of our link-joint test approach and that I expect will be used frequently, such as:

- making all *go* signals low between test operations
- making all *go* signals high for normal, self-timed, circuit operations.

To enable overlapping-high scan shift clocks as shown in the SPICE simulation waveforms in Figure 34 of Appendix F, I made the following design changes in the JTAG Controller:

1. In the Instruction Register (IR), I added a ninth register bit that is used as overlap_enable signal in the Scan Control Group and that determines whether or not the scan shift clocks can be high at the same time. IR register bits are programmed through the standard JTAG IO pins TMS and TDI — see Section 3 in Appendix F.

2. In the Scan Control Group, I added an extra mode to the Scan Control Clock Generator which enables both scan shift clocks to be high when the IR overlap_enable signal is high — see Figure 23 in Appendix F.

Overlapping-high scan shift clocks are necessary but not sufficient to support a quick reset mode at power-up. For a quick reset at power-up, we want the scan shift clocks and the scan write clock to be high and we want the scan read clock and the scan input signal to be low, for all scan chains — or at the very least for the scan chains connected to *go* and full-empty statewire signals.

To support such a quick reset, I propose to add a Scan Reset Mux for every scan chain controlled by the Scan Control Group, as shown in Figure 6.3.1. The Scan Reset Mux is inserted in each Scan Control module — see Figure 20 in Appendix F. The schematic of the Scan Reset Mux follows in Figure 6.3.2.

During power-up, we keep JTAG IO signal *TRSTb* low. As soon as the power comes up, this will cause *TRST* to go high. With *TRST* high, the Scan Reset Mux will connect the scan input signal to GND (low) instead of TDI, connect one scan shift clock to VDD (high) instead of *CK1*, connect the other scan shift clock to VDD (high) instead of *CK2*, connect the write clock to VDD (high) instead of *WR*, and connect the read clock to GND (low) instead of *RD*. This subsequently causes the low scan input to ripple through each scan chain to all *go* and full-empty signals. After power-up, we can make *TRSTb* high again, and use the normal JTAG central signals to program the scan operations or run the circuit in self-timed mode.

A complete explanation of the JTAG Controller and the changes I made to the original implementation of the Sun-Oracle Laboratories version can be found in Appendix F.

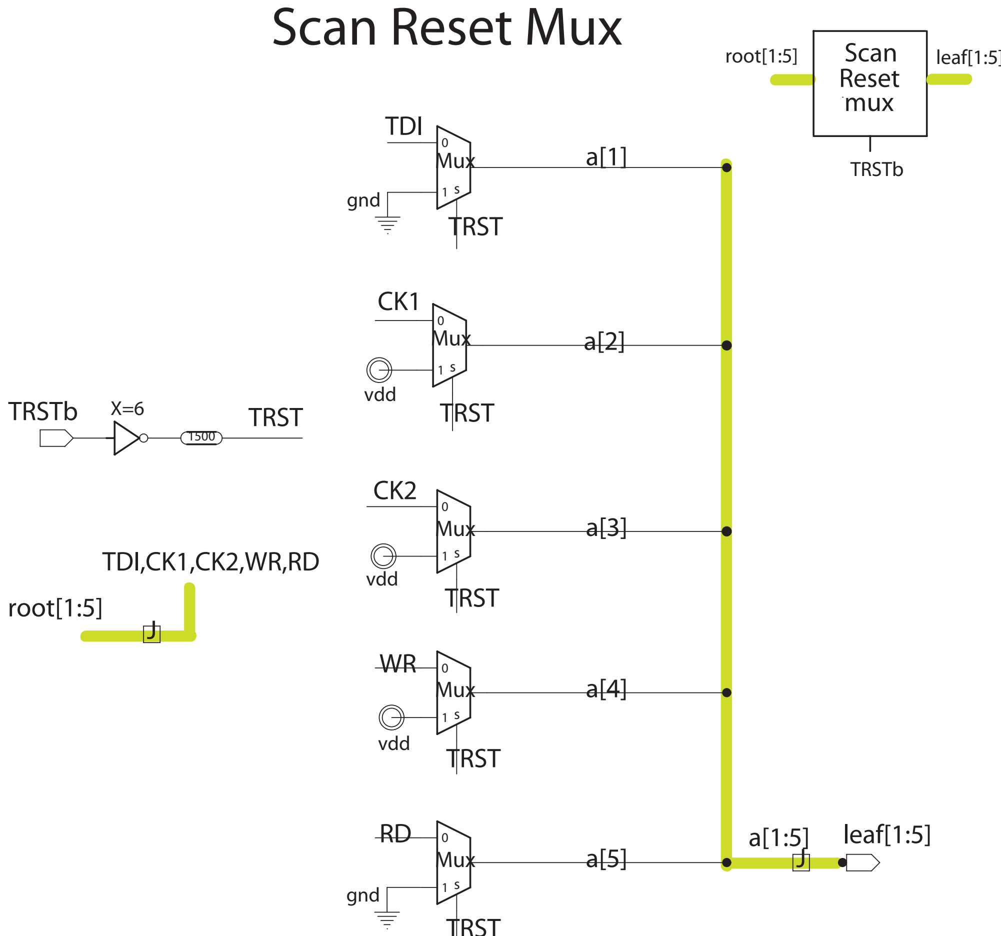


Fig. 6.3.2: Scan Reset Mux module to support initialization at power-up. This module is inserted between the JTAG Controller and the scan chains. During power-up, with *TRSTb* low, the Scan Reset Mux module makes the scan input low, makes the scan shift clocks and write clock high, and it makes the read clock low. This causes the low scan input signal to ripple through the scan chain and write a low value into all circuit signals connected to the scan chain. When *TRSTb* is high, the scan chain signals are under JTAG control.

6.4 Summary and Conclusion

The test strategy outlined in this chapter has a research as well as a practical aspect. The research aspect is in the module-level design for test solution presented in Section 6.1. The remaining Sections 6.2 and 6.3 cover the more pragmatic aspects of taking the module-level design for test solution and integrating it in an industry-standard framework.

At the module level, I added, proper start-stop control, by adding *go* signals. Combined with initialization, each *go* signal can be used as a proper start and as a proper stop signal. The resulting control circuit, called MrGO, is ANDed into each joint action – see Chapter 5. Further background and details on this can be found in ARC reports [57, 47] and in Appendix E.

At the design level, I could reuse much of the scan design features used by Sun-Oracle Laboratories to test and debug earlier chip designs in GasP. I added new scan features to control and observe each individual *go* signal and each individual GasP full-empty statewire signal. These additional scan features significantly increased the density in controllability and observability compared to the original Sun-Oracle designs, and asked for new logical and layout configurations in the scan designs. At the ARC, we revised the Electric design libraries to create scan schematics and layout solutions best suited to the new link-joint-based GasP designs with built-in testability. Some of these revisions are reported in Appendix E.

At the chip level, I added an extra test mode setting to enable initialization during power-up and to allow overlapping scan shift clocks to speed up operations that I expect will be used frequently during test and debug. Further details can be found in Appendix F.

With the help of Sun-Oracle Laboratories, the ARC manufactured two working chip experiments: Weaver and Anvil. Both experiments use the design for test solution described in Chapter 5 and Section 6.1, as well as the scan chains described in Section 6.2, and a JTAG Controller interface. Because we were not sure how the existing Sun-Oracle test software would cope with my JTAG modifications, the two chip experiments use the original Sun-Oracle JTAG Controller instead of the modified JTAG Controller described in Section 6.3 and Appendix F.

Information on the Weaver design can be found in a separate ARC report [56], downloadable from the ARC website [1].

The ARC's Weaver report is silent about how good the design for test solution is for manufacturing testing: a handful of chips won't give us the Defects-Per-Million (DPM) numbers that are required to qualify a structural test method. But ignoring DPM numbers, the Weaver and Anvil experiments described in the ARC's Weaver report and in Chapter 5 are clear evidence of a successful design for test approach and implementation.

6.5 My Contributions for Chapter 6

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

- Many of the scan test features discussed here already existed in a related form at Sun-Oracle Laboratories, where they were used to test and debug test chips with earlier GasP circuits. But this is the first time a GasP design has an externally controllable *go* signal, multiple externally controllable and observable full-empty statewires, and more than one possible initial state.
- I added an extra test mode setting to allow overlapping scan shift clocks, to speed up operations that I expect will be used frequently in our test approach, such as disabling or enabling **all** *go* signals to stop or start **all** distributed actions.
- I proposed a quick reset solution that can be activated through the existing industry-standard JTAG *TRSTb* pin, and that we can use at power-up to initialize the circuit as soon as the power comes up. This solution is external to the JTAG TAP controller — by necessity, as we cannot step the TAP controller during power-up.

7 Arbitration

A good MrGO implementation requires a good arbiter design. I have done extensive research on arbiter designs. Starting with an arbiter design presented in the 2010 ARC report ARC2010-is49 by Ivan Sutherland [52, 53], I developed:

1. a series of arbiter designs with reduced load and load variation, as seen by the input signals of the arbiter – see Appendix G,
2. a noise-robust arbiter design, by adding keepers that prevent the outputs from drifting during a contested or decided arbitration when both arbiter inputs request the arbiter's service – see Appendix H, and
3. a mathematical foundation to size arbiters for minimum uncontested grant delay.

This chapter discusses item 3. It continues with a noise-robust arbiter version, developed under item 2, henceforth called *ARC arbiter*, and formulates how to size the transistors in this version to minimize uncontested grant delays. It presents a similar formulation to size another arbiter, here called *Sparsø-Furber — or SF — arbiter*, commonly used outside the ARC. It ends by comparing the uncontested grant delays of the two arbiters.

7.1 A Tale of Two Arbiters

"First Come First Serve" (FCFS) is an important operation in many systems. An FCFS circuit, or *arbiter*, serves as an umpire to choose which of two competing input events arrived first. A key feature of an arbiter is how it treats a tie, when both inputs arrive at the same time.

Because a clocked system treats time in discrete quanta, also known as *clock periods*, "*at the same time*" in a clocked system means "*within the same clock period*," making tie resolution simple. After detecting a tie, synchronous arbiter can pick the input with highest priority or use some other simple rule to make a good choice.

Self-timed systems however treat time as a continuous rather than a quantized variable. Deciding which of two events happened first in continuous time is more difficult then in quantized time, because the difference in the arrival times may be arbitrarily small. Chaney and Molnar's 1973 classic paper [9] pointed out that, when given conflicting inputs at very nearly the same time, a flipflop might enter an intermediate state neither flipped nor flopped. The intermediate state is metastable and will eventually resolve into one of the two stable states. On rare occasions metastability may persist for a surprisingly long time. How long metastability persists depends on the flipflop's design and the proximity of the two arrivals.

Prior to the end of metastability, random noise may cause the flipflop to evolve towards either of its stable states. Fortunately it is relatively easy to use the difference in its two output voltages to tell when a flipflop is no longer metastable. Given sufficient difference between its two output voltages, a flipflop will reliably evolve towards the state that magnifies the voltage difference. Although it is

impossible to tell when metastability begins, it is easy to tell when metastability ends.

In Chapter 7 of the 1980 Mead-Conway text [20], Chuck Seitz published a simple and dependable arbiter that he called a *Mutual Exclusion* or *ME* circuit. The Seitz Mutual Exclusion circuit is dependable because it postpones its answer until after metastability, if any, ends. Seitz's circuit uses a transistor threshold voltage to detect the end of metastability. It postpones its choice until one of two ”anti-metastability” transistors begins to conduct. Which of these two transistors begins to conduct indicates reliably that the flipflop will settle in the corresponding state. Seitz's circuit is simple, elegant and dependable. Variations of Seitz's circuit have been published in several places and are widely used, including in our own work at the Asynchronous Research Center at Portland State University.

There are four CMOS versions of the original Seitz's circuit. CMOS can cross couple either NAND or NOR gates and can use either N-type of P-type transistors to detect the end of metastability. At the ARC we use a variant of Seitz's arbiter that uses cross-coupled NAND gates and N-type detection transistors as shown in Figure 7.1.1. Figure 5.20 of Chapter 5.8 of the Sparsø-Furber book [50] offers cross-coupled NAND gates and P-type detection transistors, yielding a two-stage design that closely matches Seitz's original. Figure 7.1.2 presents the topology of the Sparsø-Furber variant. Reference [50], like other texts that offer versions of the Seitz arbiter, shows the circuit topology but fails to offer guidance about what transistor sizes might offer minimum delay. Like many other publications, [50] leaves to users the choice of suitable transistor sizes.

This chapter offers help in picking transistor sizes. We consider both the *ARC*

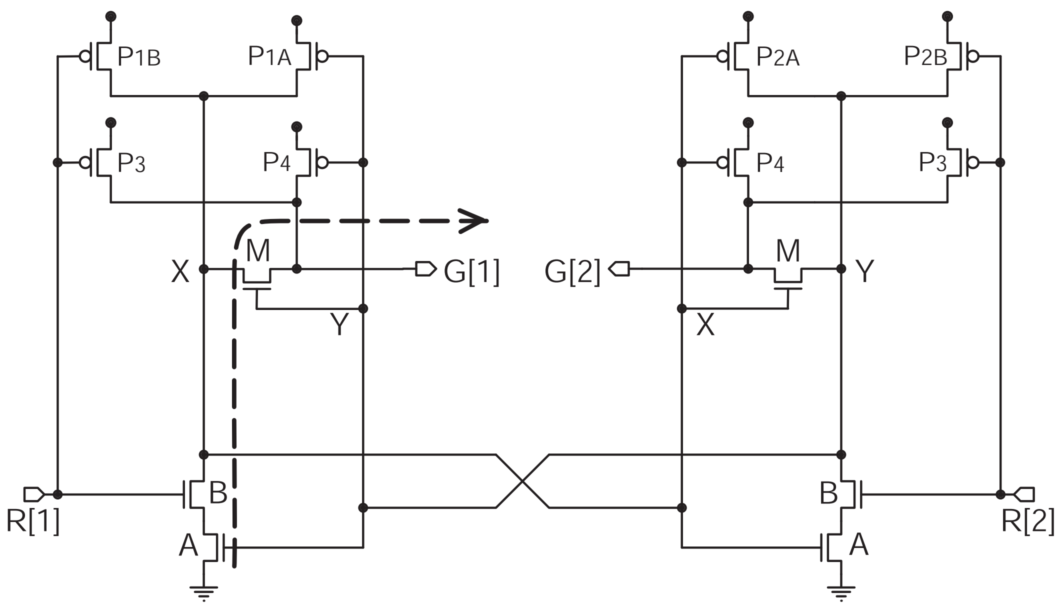


Fig. 7.1.1: The ARC arbiter. The dashed arrow shows an uncontested grant delay path from R[1] going high to G[1] going low. Node X is connected to the source of transistor M that drives G[1]. Node Y is connected to the gate of transistor M that drives G[1]. Transistors A, B and M on the left are the active transistors in the delay path from R[1] to G[1].

arbiter of Figure 7.1.1 and the *Sparsø-Furber arbiter* of Figure 7.1.2, henceforth abbreviated as *SF arbiter*. The two circuits differ in the type of transistors used to detect the end of metastability. The SF arbiter of Figure 7.1.2 is faithful to Seitz's original design, and uses the threshold voltage of P-type transistors. In contrast, the ARC arbiter of Figure 7.1.1 uses the threshold voltage of N-type transistors instead. The ARC arbiter is the result of a transcription error made many years ago by Ivan Sutherland in converting Seitz's original circuit to CMOS, and is based on Figure 5 of Appendix G.

In addition to the type of transistors used to detect the end of metastability, the two circuits are profoundly different logic-wise. The ARC arbiter of Figure 7.1.1 is an inverting one-stage design. The SF arbiter of Figure 7.1.2 is a non-inverting two-stage design.

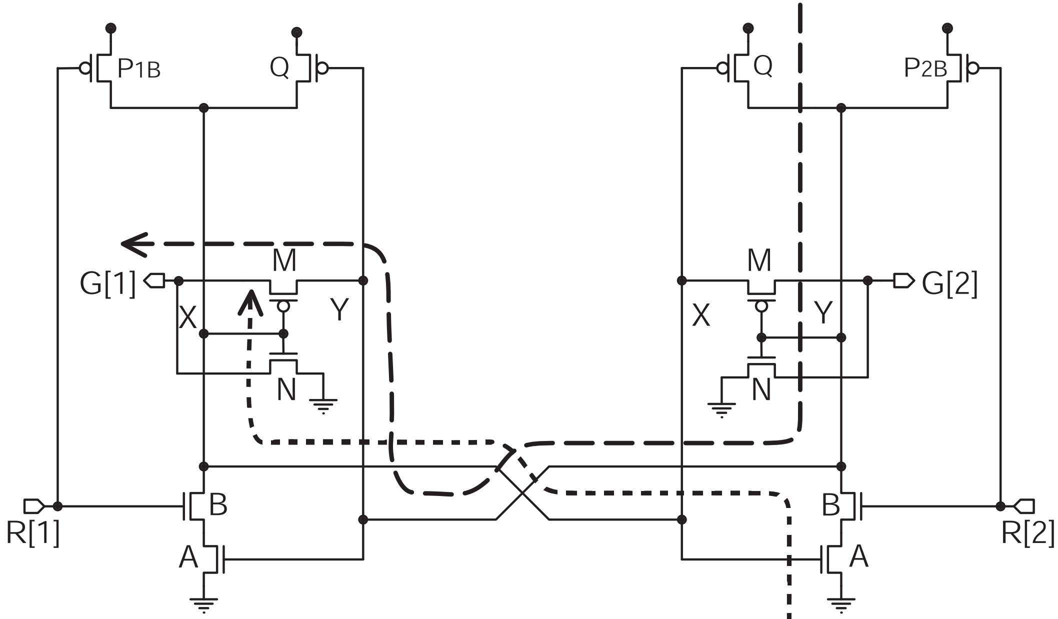


Fig. 7.1.2: The Sparsø-Furber or SF arbiter. The two dashed arrows show the uncontested grant delay path from R[1] going high to G[1] going high. Node X is connected to the gate of transistor M that drives G[1]. Node Y is connected to the source of transistor M that drives G[1]. Transistors Q, A, and B on the right and transistor M on the left are the active transistors in the delay path from R[1] to G[1].

7.1.1 How the Arbiter Circuits Work

Let us contrast the uncontested operation of the two circuits. The ARC arbiter uses three-input NAND gates, devoting the third input to detecting the end of metastability. The SF arbiter uses a separate inverting logic stage to detect the end of metastability. This difference changes the function of the anti-metastability transistors, labeled M in both Figures 7.1.1 and Figure 7.1.2. As one of the NAND gate outputs, labeled X or Y, changes, the ARC arbiter changes the source voltage of transistor M, holding M's gate voltage constant. The SF arbiter, in contrast, changes the gate voltage of transistor M, holding its source voltage constant. As illustrated by the arrows, output current from the ARC arbiter flows through N-type transistors A, B and M in series as shown

in the dashed arrow in Figure 7.1.1. Output current from the SF arbiter flows through P-type transistors Q in series with M as shown by the long-dash arrow in Figure 7.1.2.

Because the ARC arbiter has only a single stage, rising request signals R[1] or R[2] produce falling grant signals G[1] or G[2]. In contrast, the SF arbiter has two stages, and so rising request signals produce rising grant signals. Both arbiters provide mutual exclusion and so deliver at most one grant signal per arbitration.

7.1.2 Least Uncontested Delay

We seek least uncontested delay for two reasons. First, in our systems metastability is rare because the inputs to an arbiter circuit must be exceedingly close together in time to cause metastability. We optimize the much more common case of uncontested arrival. Second, in the rare case of metastability, our self-timed systems are able to wait as long as it takes for metastability to resolve. Our self-timed systems need a decisive answer but have no deadline to meet. The analysis we offer here contributes nothing to the design of synchronizer circuits used to cross clock domain boundaries. Synchronizer circuits must minimize the probability that metastability persists longer than a clock period.

To a large degree, we also ignore reset time. We are interested only in how long it takes from the rising edge of a request to the arbiter's response. How long it takes to reset the arbiter for its next decision is of lesser importance. As long as other parts of the system limit the minimum cycle time, the actual reset time is of little importance. In effect, by lengthening reset time to shorten grant time, we "*rob Peter to pay Paul*."

7.1.3 Our Simulation and Analysis

We have done both simulation and analysis to understand the impact of different transistor sizes on the performance of these two circuits. Simulation using SPICE for a variety of operating conditions shows that good performance is possible with a wide variety of transistor sizes. Mathematical analysis of circuit models yields "*optimum*" transistor sizes. We used two circuit models for our analysis of each circuit and found optimum sizes by setting to zero a partial derivative of delay with respect to transistor size. The "*optimum*" transistor sizes found in this way are well within the range of good sizes suggested by SPICE. The simulations make clear that sizes close to optimum provide nearly as good performance. It is encouraging to know that the two circuits tolerate size variation well.

The mathematical analysis reveals useful relationships between the sizes of various transistors. The analysis shows how sizes depend on the amount of drive expected of the circuits.

The mathematical analysis results are discussed in Section 7.2.

The SPICE simulation results are discussed in Section 7.3

7.2 Mathematical Analysis

Our strategy for analysis starts with a delay equation for the uncontested case. We treat total delay as the sum of small delays for each part of the circuit. Although some of these delays overlap in time, the sum seems appropriate because each driving transistor must provide charge to each of its parallel loads.

Picking which parts of the circuit to represent requires careful examination of the structure of each circuit.

We model the arbiter circuits with a similar input to output stepup for each. Given an arbitrary fixed size for the input transistor, labeled *B* in both Figure 7.1.1 and Figure 7.1.2, the load imposed on each arbiter circuit is s times bigger, where s is the desired stepup.

We use upper-case letters to represent transistors and lower-case letters to represent drive strengths. Notice that drive strength differs from transistor width. To achieve similar strength, P-type transistors must be wider than N-type transistors.

In our mathematical analysis, we will consider the ARC and SF arbiters from a parametric point of view. **The hard question is what parameters to adopt.**

7.2.1 Parameters: Load, Strength, Delay

We can measure and express delay in explicit time units, like picoseconds, or in-explicit units, relative to a reference gate delay. For our delay analysis, we have taken the latter approach, using relative gate delays. We express absolute delay, D_{abs}, as the product of (1) total gate delay units, *D*, and (2) the reference gate delay, "tau," also written as τ:

$$D_{abs} = D * \tau \tag{7.2.1}$$

Parameter *D* is unitless and process independent. The process is encoded in τ. Specifically, τ is the delay of an inverter, without parasitic loads, driving an identical inverter. In summary, we compute delay in terms of *D*, i.e., in terms of total gate delay units, and scale the result to the proper circuit process, using τ.

For each part of the circuit, *D* can be computed as the ratio of (1) output load for this part of the circuit to (2) drive strength of this part of the circuit. For background information, see the 1999 "Logical Effort " book by Ivan Sutherland et al. [58] and the 2004 publication on "Transistor Sizing" by Jo Ebergen et al. [11]:

$$D = \frac{Load}{Strength} \tag{7.2.2}$$

Load is the sum of all the capacitances at the output of this circuit part. Strength is a measure of how much charge the part can deliver per unit time. We assume that all transistors are of minimum length, so a transistor's strength is proportional to its width, *w*, as are the capacitance of its gate and its ability to produce an output current. Under these assumptions, we may consider the size of a transistor to be its width.

The *size* of a transistor differs from the transistor's *strength*, as follows. If an N-type transistor of strength 1 presents an input capacitance of C_N to its driver, then a P-type transistor of strength 1 presents an input capacitance of $\gamma * C_N$. The constant γ, expressed by the Greek letter "gamma," expresses how much bigger a P-type transistor must be compared to an N-type transistor to deliver the same charge. A γ times bigger, i.e., wider, P-type transistor is also γ times harder to drive. Constant γ is process dependent, and generally greater than 1. It was often taken to be 2 in older fabrication processes, but its value in modern processes tends to be closer to 1.

Load is expressed in units matching the reference input capacitance, C_{ref}, presented by the smallest size inverter available in the given process technology, with one strength-1 N-type transistor and one strength-1 P-type transistor, where:

$$C_{ref} = C_N + (\gamma * C_N) = (1 + \gamma) * C_N \; .$$

Thus, relative to C_{ref}, the input capacitance of a strength-1 N-type transistor with input capacitance C_N is:

$$\frac{C_N}{(1+\gamma)*C_N} = \frac{1}{(1+\gamma)}.$$

Likewise, relative to C_{ref}, the input capacitance of a strength-1 P-type transistor with input capacitance $\gamma * C_N$ is:

$$\frac{C_N}{(1+\gamma)*\gamma*C_N} = \frac{\gamma}{(1+\gamma)}.$$

Table 7.2.1 generalizes these formulas to transistors of strength x, for $x \geq 1$.

For the delay analysis of the arbiters in Figures 7.1.1 and 7.1.2, we will need to know the drive strength of two or three transistors in series. For instance, transistor A is connected in series with transistor B.

We can express the drive strength of such serial configurations in a normalized form. We can define $s(a,b)$, the normalized drive strength of transistor A with

Table 7.2.1: P-type and N-type transistor equations for input capacitance versus strength, assuming P-type transistors are γ times harder to drive than N-type transistors. We assume that the input capacitances on a transistor's gate, drain, and source are similar, as is the case in the 90nm and 40nm manufacturing processes we have worked with.

Type	Strength	Input capacitance of transistor's Gate or Drain or Source
N	x	$\frac{x}{1+\gamma}$
P	x	$\frac{\gamma*x}{1+\gamma}$

strength a in series with transistor B with strength b, as the reciprocal of a sum of reciprocals, as follows:

$$s(a,b) = \frac{1}{\frac{1}{a}+\frac{1}{b}} = \frac{a*b}{a+b} \tag{7.2.3}$$

Similarly, we can define $s(a,b,m)$, the normalized drive strength of the three in-series transistors A, with strength a, B, with strength b, and M, with strength m, as follows:

$$s(a,b,m) = \frac{1}{\frac{1}{a}+\frac{1}{b}+\frac{1}{m}} = \frac{a*b*m}{(a*m)+(b*m)+(a*b)} \tag{7.2.4}$$

Best transistor strengths are a strong function of stepup. To aid comparison of the two arbiter circuits in Figures 7.1.1 and 7.1.2, we choose an identical input capacitance for each. The R[1] input load for the uncontested R[1] to G[1] path in the ARC arbiter, shown in Figure 7.1.1, is presented only by transistor B, because we ignore the tiny P-type reset transistors.

Likewise, the R[1] input load for the uncontested R[1] to G[1] paths in the SF arbiter, shown in Figure 7.1.2, is presented also only by transistor B, because we ignore the tiny P-type reset transistor, P1B. For each arbiter analysis we will have the arbiter drive a load that is s times as big as its input load, where s is the stepup expected of the arbiter. In practice we tend to load arbiters lightly by keeping s small, e.g., 2 or 3.

The best strength for transistor M is a compromise between (1) the load M presents to the NAND gates that drive it and (2) M's ability to drive the arbiter output. A stronger M speeds driving the arbiter's output, but a stronger M is also harder to drive and therefore retards action of the NAND gates. Although a weaker M is easier for the NAND gates to drive, a weaker M takes longer to drive the arbiter output. Analysis of the delay equations for the R[1] to G[1] paths shows the best compromise. Likewise, the best strengths for transistors other than M is a compromise between their ability to drive loads and the load they impose on their inputs. For instance transistor A drives B and M in series and is driven by the other A and B in series.

Our goal to seek the least uncontested delay translates to seeking the best strengths for transistors A and M in relation to B.

7.2.2 Overall Delay Analysis

We analyzed each arbiter's uncontested request-grant delay, for R[1] to G[1], using two optimization methods — the first without and the second with transistor parasitic loads. Both methods use only transistors that play an active role in the request-grant path. We represent the delays in τ units — see Section 7.2.1.

The two optimization methods differ in the absolute delay values that they produce for each arbiter. The two methods grossly agree, though, on how to size the transistors A and M relative to B to minimize the least uncontested delay — which is the result that we seek. This chapter will discuss only the non-parasitic optimization method. Details about the parametric analysis are left for a future publication.

7.2.3 ARC Arbiter Analysis

In this section, we analyze the ARC arbiter. We want to find the best strengths for transistors A and M for minimizing the total uncontested delay of the ARC arbiter in Figure 7.1.1, as represented by the uncontested delay from R[1] going high to G[1] going low.

The best strengths depend on the strength of B and on how much load the arbiter drives relative to B. We use lower case variables a, b, and m, to denote the strengths of transistors A, B, and M, respectively. We will generate answers in terms of how much stronger or weaker than transistor B should be transistors A and M. For our calculations we will ignore the capacitive loads of transistors P1B, P1A, P2A, P2B, P3, and P4, because these keeper transistors play an active role only in the reset phase, and are small enough to have only a marginal effect on our results.

Note that making A or M stronger both expedites and retards the uncontested path:

- A stronger A or M expedites the path by improving strengths $s(a, b)$ and $s(a, b, m)$.
- A stronger A or M retards the path by increasing the load on the NAND gates.

Our plan is first to write an equation for delay in terms of our variables, a and m. Armed with such an equation we can find a best value for each variable by equating to zero the partial derivative of the delay with respect to that variable.

We will consider the total delay as partitioned serially over three sub-paths.

Each sub-path will show a combination of transistors driving an individual load. We will express the total delay as the sum of the three smaller sub-path delays as if the sub-path delays were separate. In reality, the drives overlap in time. However, the sum is valid because the time it takes to fill a combination of loads with charge is the sum of the times it would take to fill with charge each individual load.

For the first delay, $D1$, we consider the other A and B filling A with charge. For the second delay, $D2$, we consider A and B filling M with charge. For the third delay, $D3$, we consider A, B, and M filling the final output load $s * b$ on G[1] with charge.

From equation 7.2.2 and Table 7.2.1 we derive:

$$D1 = \frac{a}{(1+\gamma) * s\,(a,b)} \quad \text{(note [1])}$$

$$D2 = \frac{2 * m}{(1+\gamma) * s\,(a,b)} \quad \text{(note [2])}$$

$$D3 = \frac{s * b}{(1+\gamma) * s\,(a,b,m)}$$

$$Dtotal_{ARC} = D1 + D2 + D3$$

Using equations 7.2.3–7.2.4, the resulting equation for total delay is given by:

$$Dtotal_{ARC} = \frac{(a+b)(a+(2*m))}{(1+\gamma)*a*b} + \frac{s*((a*b)+(a*m)+(b*m))}{(1+\gamma)*a*m} \tag{7.2.5}$$

[1] Remember that we ignore keeper transistor loads.

[2] Note that A and B in Figure 7.1.1 drive the source of one M and the gate of the other M transistor.

Table 7.2.2: Equations for best strengths for transistors A and M in the ARC arbiter of Figure 7.1.1 to minimize the least uncontested request to grant delay — ignoring parasitics.

ARC arbiter's best strength for A	$a = \sqrt{(b * ((2 * m) + (s * b)))}$
ARC arbiter's best strength for M	$m = \sqrt{(\frac{a * b}{(a + b)} * \frac{s * b}{2})}$

We found a best strength value for each of the variables a and m by equating equation 7.2.5 so as to zero the partial derivative of the delay with respect to the variable. The results follow in Table 7.2.2.

7.2.4 Sparsø-Furber (SF) Arbiter Analysis

In this section, we analyze the Sparsø-Furber, or SF, arbiter. We want to find the best strengths for transistors A, M, and Q for minimizing the total uncontested delay of the SF arbiter in Figure 7.1.2, as represented by the uncontested delay from R[1] going high to G[1] going high.

The best strengths depend on the strength of B and on how much load the SF arbiter drives relative to B. We use the lower case variables, a, b, m, and q, to denote the strengths for transistors A, B, M, and Q. We will generate answers in terms of how much stronger or weaker than transistor B transistors A, M, and Q should be. In this analysis, we ignore the load of keeper transistors P1B, P2B, and N.

Following the approach in Section 7.2.3, our plan is first to write an equation for delay in terms of our variables, a, m, and q. Armed with such an equation we

can find a best value for each variable by equating to zero the partial derivative of the delay with respect to that variable.

We will consider the total delay as partitioned serially over two sub-paths. Each sub-path will show a combination of transistors driving an individual load. We will express the total delay as the sum of the two smaller sub-path delays as if the delays were separate. In reality, the drives overlap in time. Nevertheless, the sum is valid because the time it takes to fill a combination of loads with charge is the sum of the times it would take to fill with charge each individual load.

For the first delay, $D1$, we consider A and B filling the other A as well as M and Q with charge — following the short-dash arrow in Figure 7.1.2. For the second delay, $D2$, we consider M and Q filling the final output load $s * b$ on G[1] with charge — following the long-dash arrow in Figure 7.1.2.

From equation 7.2.2 and Table 7.2.1 we derive:

$$D1 = \frac{a + (2 * \gamma * m) + (2 * \gamma * q)}{(1 + \gamma) * s\,(a, b)} \quad \text{(note [3])}$$

$$D2 = \frac{s * b}{(1 + \gamma) * s\,(m, q)} \quad \text{(note [4])}$$

$$Dtotal_{SF} = D1 + D2$$

[3]Note that A and B in Figure 7.1.2 drive the source of one M and the gate of the other M transistor.

[4]Note that A and B in Figure 7.1.2 drive the drain of one Q and the gate of the other Q transistor.

Table 7.2.3: Equations for best strengths for transistors A, M, and Q in the SF arbiter of Figure 7.1.2 to minimize the least uncontested request to grant delay — ignoring parasitics. We assume that P-type transistors are γ times harder to drive than N-type transistors — see Section 7.2.1.

SF arbiter's best strength for A	$a = \sqrt{(b * \gamma * ((2 * m) + (2 * q)))}$
SF arbiter's best strength for M	$m = \sqrt{(\frac{a * b}{(a + b)} * \frac{s * b}{2 * \gamma})}$
SF arbiter's best strength for Q	$q = \sqrt{(\frac{a * b}{(a + b)} * \frac{s * b}{2 * \gamma})}$

Using equation 7.2.3, the resulting equation for total delay is given by:

$$Dtotal_{SF} = \frac{(a + b)(a + (2 * \gamma * m) + (2 * \gamma * q))}{(1 + \gamma) * a * b} + \frac{s * b * (m + q)}{(1 + \gamma) * m * q} \qquad (7.2.6)$$

We found a best value for each of the variables a, m, and q, by equating equation 7.2.6 so as to zero the partial derivative of the delay with respect to the variable. The results follow in Table 7.2.3.

7.2.5 Mathematical Delay Analysis: Summary and Comparison

Tables 7.2.2–7.2.3 summarize the results for minimizing delay equations 7.2.5 and 7.2.6 through variables a, m, and q.

Note that for the ARC arbiter, the best strength a for transistor A is the geometric mean[5] of (1) the strength of B, b, and (2) the load driven by the transistor-AND series A-B: two M transistors, each of strength m, and the scaled output load $s * b$.

[5]The geometric mean of X and Y is the square root of their product: $\sqrt{(X * Y)}$.

Quite similarly, for the SF arbiter, the best strength a for transistor A is the geometric mean of (1) the strength of B, b, and (2) the load driven by transistor-AND series A-B: two M transistors, each of strength m, and two Q transistors, each of strength q. In the SF arbiter, the best strength of A is independent of the scaled output load. This is because the SF arbiter has two subsequent stages. Transistor A plays a role only in the first stage that drives X, as indicated by the small-dash arrow in Figure 7.1.2.

For both arbiters, the best strength m for transistor M is approximately the geometric mean of the transistor-AND series A-B and the scaled output load $s * b$.

For the SF arbiter, the best strength q for transistor Q is approximately the geometric mean of the series transistors AND pair and the scaled output load $s * b$. Transistors M and Q in the SF arbiter have the same sizing equations. For least uncontested delay, the strength for m and q are equal. Also note that the term γ appears in all SF arbiter equations. This is because M and Q are P-type transistors.

To compare $Dtotal_{ARC}$ versus $Dtotal_{SF}$ for different stepups s, we will keep strength b of transistor B constant, thus providing a common reference point for input capacitance, b, and a common reference point for output load, $s * b$, per stepup.

With b fixed to 12 and stepup s fixed to first 1, then 2, and finally 3, we can now solve the equations in Tables 7.2.2 and 7.2.3 — e.g. by using MATLAB [4] — to get the best strength a, m, and q for each arbiter and for each stepup. Substituting these best strengths into equations 7.2.5 respectively 7.2.6 gives us each arbiter's least uncontested grant delay, $Dtotal_{ARC}$ respectively $Dtotal_{SF}$, for each stepup. The results follow in Table 7.2.4.

The last column in Table 7.2.4 indicates that the ARC arbiter has a smaller, i.e., better, least uncontested grant delay than the SF arbiter. The ARC arbiter's least uncontested grant delay is 32–39% better.

Remember, though, that the mathematical analysis presented in this chapter ignores parasitic loads — see Section 7.2.2. In the next section, we will look at SPICE simulations, which include parasitic loads, and compare the two arbiters again in SPICE.

Table 7.2.4: Mathematically best strengths and uncontested grant delays for both arbiters. The best strengths a, m, and q for transistors A, M and Q are based on Tables 7.2.2 and 7.2.3, with b fixed to 12, and γ fixed to 2. Least uncontested grant delay, D, expressed in τ units, is based on delay equations 7.2.5, for the ARC arbiter, and 7.2.6, for the SF arbiter. The decimal numbers for a, m, q, and D have been rounded to the nearest tenth.

	Stepup	Best strength a of A	Best strength m (q) of M (Q)	D [τ] (% over SF)
ARC Arbiter	1	17.3	6.5	**2.6** (-39%)
	2	22.8	9.7	**3.6** (-36%)
	3	26.9	12.0	**4.5** (-32%)
SF Arbiter	1	22.8	5.0	**4.3**
	2	25.0	7.0	**5.6**
	3	26.5	8.6	**6.6**

7.3 SPICE Delay Simulations

To complement the mathematical analysis presented in Section 7.2, I will redo the sizing for best transistor strengths and least uncontested grant delay using SPICE-based circuit simulations [38] in a 90nm CMOS process technology. As in Table 7.2.4, the SPICE simulations are set up with strength b for transistor B fixed to 12, with γ fixed to 2, and with stepup s fixed to first 1, then 2, and finally 3.

Because the SPICE simulations consider transistor parasitic capacitances, the strengths of transistors other than A, B, M, and Q are taken into account. For the ARC arbiter in Figure 7.5.1, I fixed the strengths for transistors P3, P1B, and P2B to 1, to reduce the input capacitance of the arbiter. I made the strength of keeper transistor P4 slightly higher, using 2. I fixed the strengths for transistors P1A and P2B to 4, to ease the arbiter's reset and release phase. Likewise, for the SF arbiter in Figure 7.5.2, I fixed the strengths for transistors P1B and P2B to 2, to reduce the input capacitance of the arbiter in a way similar to that of the ARC arbiter. I fixed the strength of keeper transistor N to 4 and the strengths for transistors P1B and P2B to 6, to ease the arbiter's reset and release phase.

To give both arbiters the same stepup, I calculated the normalized input capacitance of each arbiter, and then loaded each arbiter with an s times bigger output capacitance. The normalized input capacitance for each arbiters is determined by (1) N-type transistor B, of strength 12, and (2) two P-type transistors of strength 1 for the ARC arbiter versus one P-type transistor of strength 2 for the SF arbiter. Based on Table 7.2.1, both result in a normalized input capacitance of $(4 + \frac{4}{3}) \approx 5.33$. I have rounded off the corresponding output loads for a stepup s of 1, 2, and 3 to respectively 5.5, 11, and 16.

With B and all transistors that play a secondary role to the least uncontested grant delay fixed, I can now sweep strengths a, m, and q of transistors A, M, and Q, and measure the corresponding SPICE-simulated uncontested grant delay of each arbiter. The delays are measured from the time request input signal R[1] reaches its 50%-VDD voltage level to the time that grant output signal G[1] reaches its 50%-VDD voltage level.

The sweep results follow in Figure 7.3.1. The X-axis sweeps strength a of transistor A. The Y-axis shows the corresponding uncontested R[1] to G[1] delay in picoseconds. The six graphs plotted represent the best-strength m results for the given arbiter, the given range of a sizes, and the given stepup. For instance,

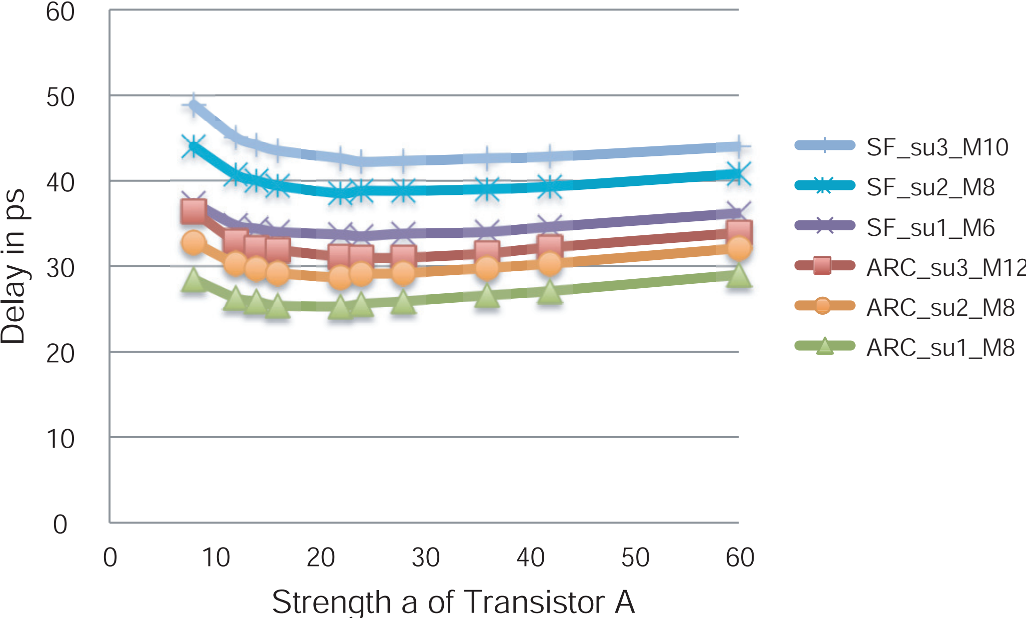


Fig. 7.3.1: SPICE-simulated delay graphs, for the two arbiters implemented in a 90nm CMOS process with $\gamma = 2$. Transistor B strength b has been fixed to 12. The X-axis sweeps strength a of transistor A. The Y-axis shows the uncontested R[1] to G[1] delay in picoseconds. The top three graphs belong to the best-m simulation for the SF arbiter for stepups 3 (highest, in grey) to 1 (lowest, in purple). The bottom three graphs belong to the best-m simulation for the ARC arbiter for stepups 3 (highest, in red) to 1 (lowest, in green).

the graph with name *ARC_su1_M8* represents the best delay graph with the lowest R[1] to G[1] delays for the ARC arbiter, as indicated by the name's prefix "ARC," simulated with a stepup of 1, as indicated by the name's middle term "su1." This best delay graph was obtained from a simulation using strength m of 8 for transistor M, as indicated by the name's suffix "M8."

As we sweep strength a of transistor A, we see a minimum delay point at the "best-a." For strengths greater or smaller than this best-a, the delay increases. The same is true in the sweeps of strength m for transistor M. This is because making A or M stronger or weaker both expedites and retards the uncontested R[1] to G[1] path — see also page 142.

The delay graphs are relatively flat around each best-a, best-m point. The arbiter delays are nearly constant for a good range of transistor A and M strengths. This gives the designer some flexibility in sizing the transistors. Looking at the SPICE simulation results for all three stepups we conclude that — relative to transistor B strength b:

- Best strength a of A is obtained around $2 * b$.
- Best strength m of M is obtained around $\frac{2}{3} * b$.

Table 7.3.1 summarizes the simulated best-a, best-m and least uncontested grant delays. The last column indicates that the ARC arbiter has a smaller, i.e., better, grant delay than the SF arbiter. The ARC arbiter's least uncontested grant delay is 23–27% better.

7.4 Comparison of Analyzed versus Simulated Results

SPICE-based ring oscillator experiments, performed for the 90nm CMOS process in which we simulated the two arbiter designs, indicate that τ is 6–8 picoseconds.

Additional experiments show that an inverter with one strength-1 N-type transistor and one strength-1 P-type transistor, driving three similar inverters, has a delay of 20 picoseconds, i.e., 2.5 to 3.3 τ units. We assume this delay to be our typical "gate delay."

The analyzed versus simulated comparisons in Table $7.4.1$ normalize all delays to τ units, using a τ unit of 7 picoseconds. As indicated in the last column, the results differ by at most one τ unit, i.e., a third of a gate delay — which we feel is good enough, considering that the analysis ignores transistor parasitic

Table 7.3.1: SPICE-simulated best strengths and uncontested grant delays. The best strengths a, m, and q for transistors A, M, and Q are based on the results presented in Figure 7.3.1. The least uncontested grant delay, D, expressed in picoseconds (ps), is the minimum delay at the best-a, best-m(and q) point of the corresponding arbiter-stepup graph.

	Stepup	Best strength a of A	Best strength m (q) of M (Q)	D [ps] (% over SF)
ARC Arbiter	1	22	8	**25.5** (-27%)
	2	22	8	**29.9** (-23%)
	3	24	12	**33.0** (-23%)
SF Arbiter	1	24	6	**34.7**
	2	22	8	**39.0**
	3	24	10	**42.6**

capacitances.

More importantly, the additional analyzed and simulated results in Tables 7.2.4–7.3.1 show trends that are similar: both indicate that the ARC arbiter's grant delay is somewhat better (32–39% versus 23–27%) and both point to similar best transistor sizing strategies.

7.5 Sizing Validation from a Design Perspective

Figures 7.5.1–7.5.2 show best-strength ARC arbiter and SF arbiter schematics, with transistor strengths agreeable with our mathematical analysis and with the SPICE simulation settings and optimization results in Sections 7.3 and Table 7.3.1.

In this section, we look at the resulting schematics and check design aspects that were temporarily put aside during our mathematical analysis and SPICE simulation.

Table 7.4.1: Analyzed versus simulated uncontested grant delays, compared with $\tau = 7\,$ps.

	Stepup	D [τ] from Table 7.2.4	D [ps] ([τ]) from Table 7.3.1	Difference in τ
ARC Arbiter	1	2.6	25.5 (3.6)	1.0
	2	3.6	29.9 (4.3)	0.7
	3	4.5	33.0 (4.7)	0.2
SF Arbiter	1	4.3	34.7 (5.0)	0.7
	2	5.6	39.0 (5.6)	0.0
	3	6.6	42.6 (6.1)	−0.5

One of those design aspects concerns each arbiter's reset time, which we ignored to a large degree, as noted at the end of Section 7.1.2:

- As long as other parts of the system limit the minimum cycle time, the actual reset time is of little importance. In effect, by lengthening reset time to shorten grant time, we "rob Peter to pay Paul."

We want to make sure that the reset time fits within the minimum cycle time of our self-timed designs — i.e., continuing the metaphor, that "Peter lives above the poverty line."

To establish their reset times, I re-simulated the two optimized arbiters in SPICE. For the ARC arbiter in Figure 7.5.1, I measured the uncontested grant delay from R[1] high to G[1] low, and I measured the reset delay from R[1] low to G[1] high. Similarly, for the SF arbiter in Figure 7.5.2, I measured the uncontested grant delay from R[1] high to G[1] high, and I measured the reset delay from R[1] low to G[1] low. As in Section 7.3, all delays are measured from and to a 50%-VDD voltage level.

Figure 7.5.3 shows the four corresponding delay graphs, with delays normalized to τ units of 7 picoseconds (as in Table 7.4.1), showing three stepup delays per graph — plus a no-load delay, extrapolated for a stepup s of 0, with the arbiter driving nothing.

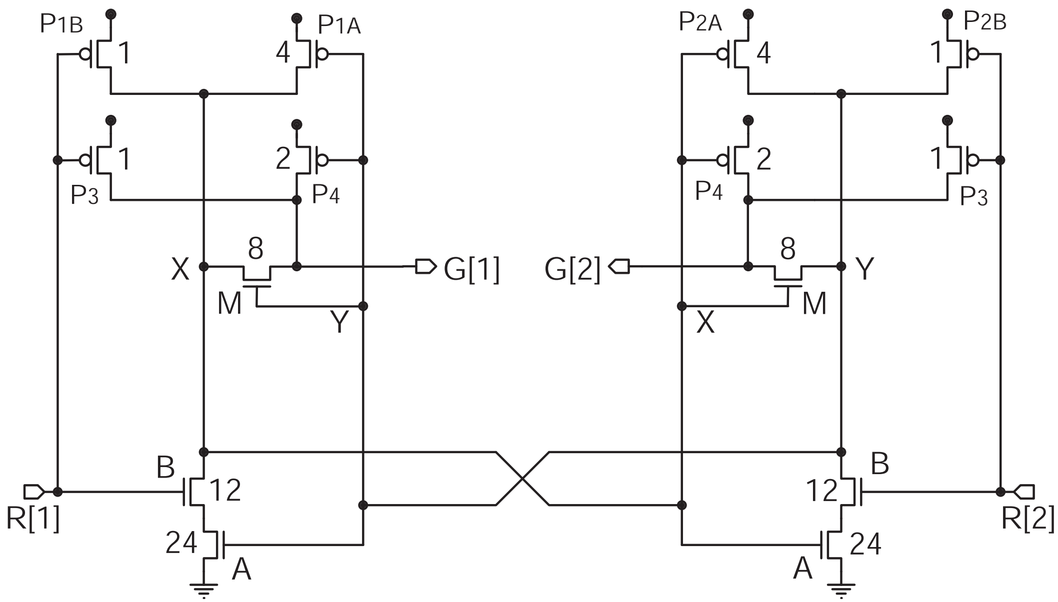


Fig. 7.5.1: ARC arbiter with optimized strengths for transistors A and M, sized relative to transistor B whose strength is fixed to 12. The results are based on our mathematical analysis and the SPICE-simulation settings and results in Section 7.3 and Table 7.3.1.

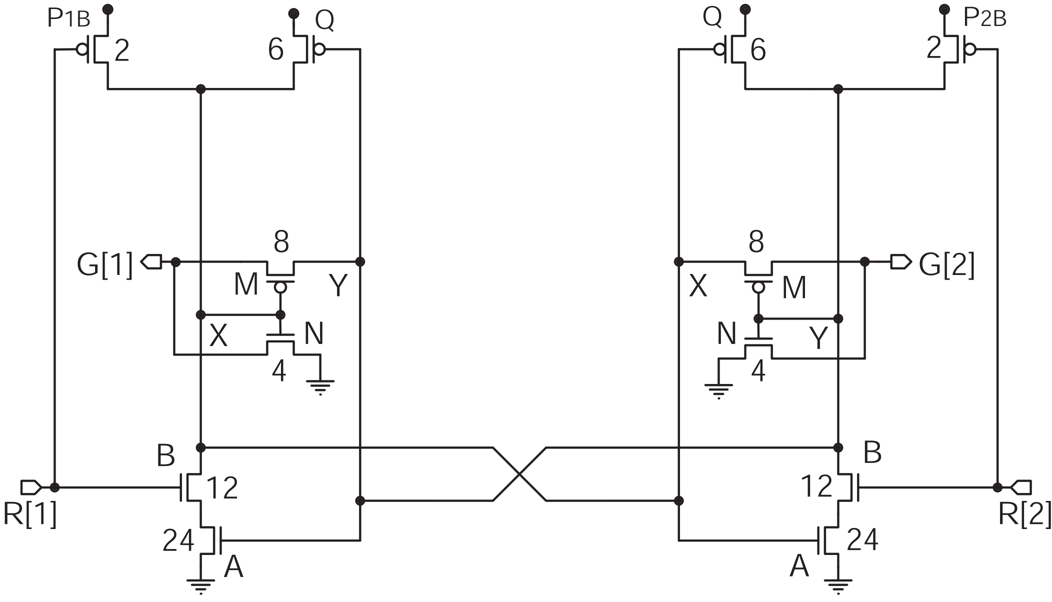


Fig. 7.5.2: SF arbiter with optimized strengths for transistors A, M, and Q, sized relative to transistor B whose strength is fixed to 12. The results are based on our mathematical analysis and the SPICE-simulation settings and results in Section 7.3 and Table 7.3.1.

As reference, I have also included delay graphs for a NAND gate and an inverter, each with equal input-to-output delays for rising and falling output.

The delay graphs in Figure 7.5.3 show three other design aspects of interest, namely: how good each design is regarding (1) performing its logical function, e.g., arbitrate, in addition to (2) driving its output signal, i.e. acting as input-to-output amplifier, as well as (3) driving its own internal capacitance. Specifically, for each gate delay, D, we have [58]:

$$D = gh + p$$

where:

- g is the logical effort,
- h is the electrical effort, here also known as stepup s, and
- p is the internal parasitic delay, i.e. the delay for $s = 0$, driving nothing.

Figure 7.5.3 shows that the logical effort numbers (g) for the two arbiters are surprisingly good. To perform uncontested grants, the logical effort of each arbiter is smaller than that of an inverter!

But the key message to take away from Figure 7.5.3 is that the maximum reset delays for the two arbiters stay below 5 gate delay, i.e., $15\ \tau$ units, which is the minimum self-reset time that we use for our 6-4 GasP and our Click designs.

So, as long as we avoid overloading the arbiters, our arbiter optimization strategy and the resulting arbiter designs in Figures 7.5.1–7.5.2 are valid from a design point of view.

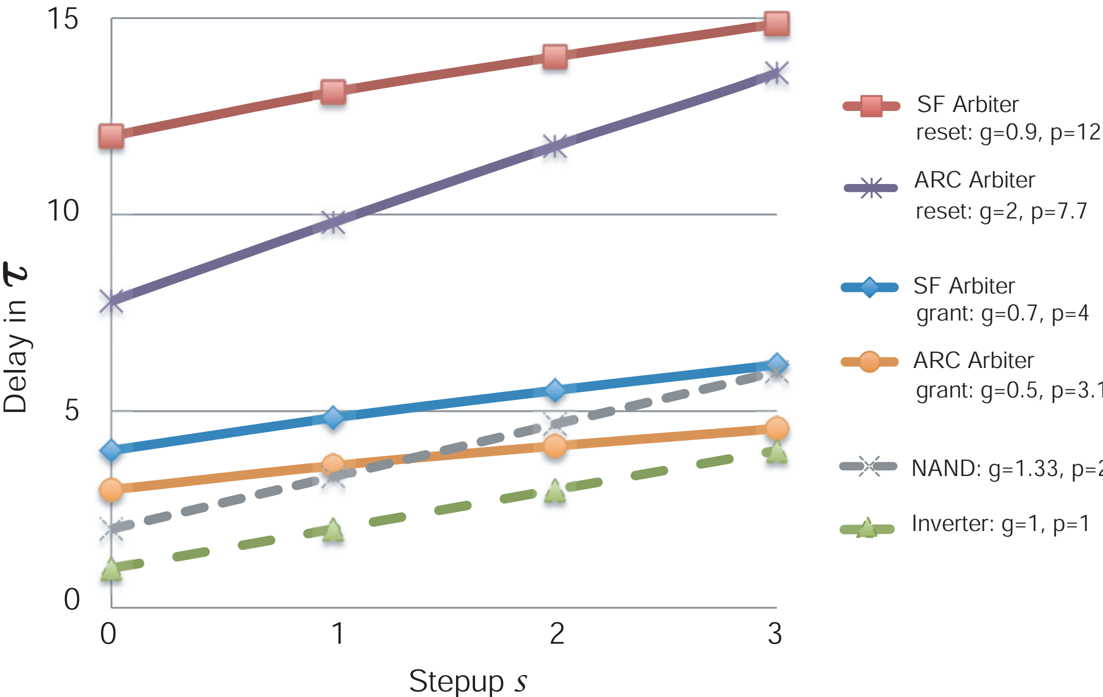


Fig. 7.5.3: Optimally sized arbiter graphs (solid line segments) for the ARC arbiter in Figure 7.5.1 and the SF arbiter in Figure 7.5.2. Shown are 90nm SPICE simulation results for the arbiter's uncontested grant and reset delays, for a stepup s of 1, 2, and 3. Each graph has its own logical effort (g) as well as a parasitic delay (p) at $s = 0$. The two extra graphs (dashed line segments) serve as reference, and show the input-to-output delays for a strength-1 NAND gate and a strength-1 inverter — again for a stepup s of 1, 2, and 3.

7.6 Summary and Conclusion

We analyzed two arbiter designs – the ARC arbiter and the Sparsø-Furber (SF) arbiter. We derived mathematical equations for transistor strengths to minimize the most important delay in each arbiter — the uncontested request-to-grant delay. Our analysis provides insight into the role played by the non-keeper transistors labeled A, B and Q in Figure 7.1.1 – Figure 7.1.2 and the metastability transistor labeled M. From the equations we derived best strengths for A, M, and Q relative to B for least uncontested grant delay.

We confirmed our equations with SPICE simulations for a 90nm CMOS manufacturing process. Like the analysis results, the SPICE simulation results indicate that the ARC arbiter has a smaller, i.e., better, uncontested grant delay than the SF arbiter, for similar input capacitance and stepup. In SPICE, the ARC arbiter's least uncontested grant delay is approximately 25% better. The SPICE simulations also indicate that each arbiter's uncontested grant delays are nearly constant for a good range around the best transistor A, M, and Q strengths. This gives the designer some flexibility in sizing the transistors. Both optimized arbiter designs have surprisingly good logical effort.

In our analytical delay analysis, we used two optimization methods — one without and one with transistor parasitic capacitances. Both optimization methods give exact sizing optima well within the range suggested by our SPICE simulations. This chapter omits the mathematical analysis with parasitic capacitances, because the extra details add little to the final results.

The final results provide the designer guidance about the best relative strengths of transistors A, B, M, and Q. If we fix the strength of transistor B, the best strength for transistor A is any value between 1.5 to 2 times the strength of

transistor B. Roughly, M is the geometric mean of the strength of A and B in series and the arbiter load. The best strength of transistor M is any value between 0.5 to 0.8 times the strength of transistor B. Transistor Q can have the same strength as transistor M. But — as the SPICE simulations show — missing these choices by a factor of 2 or so doesn't matter.

A key motivation for this arbiter analysis is the presence of an arbitrated MrGO in each and every joint – see Chapter 5. MrGO uses a variant of the ARC arbiter in Figure 7.1.1, but with only one output.

For future work, a similar analysis and simulation with arbiter designs that use cross-coupled NOR gates instead of cross-coupled NAND gates would be welcome.

7.7 My Contributions for Chapter 7

While most of this thesis is joint work with my supervisors, the following key contributions are largely mine.

- I developed a series of arbiter designs for reduced load and load variation as seen by the input signals of the arbiter. A reduced input load is important especially for arbiter inputs that are weakly driven or that are driven by keepers on potentially long wires, such as GasP full or empty indicating statewires. A reduced input load is less important for MrGO arbiter inputs, because MrGO is driven locally by a strong AND function.
- I developed a noise-robust arbiter by adding keepers that prevent the arbiter outputs from drifting during a contested or decided arbitration when both arbiter inputs request the arbiter's service. These keepers are essen-

tial for the arbiter to operate in an asynchronous environment where inputs may wait for arbitrarily long times before being served. These keepers are now an integral part of all ARC arbiters.

- I took the noise-robust arbiter that we, at the ARC, use as the basis for MrGO, and compared it to the elsewhere commonly used Sparsø-Furber (SF) arbiter. For similar input capacitive load and a corresponding stepup to output capacitive load, the ARC arbiter's uncontested request-to-grant is approximately $25\,\%$ faster.

- I presented an analytical analysis as well as SPICE simulations for the two arbiters. The final results provide the designer with guidance about how to size such arbiters — guidance that was much needed and not addressed in the existing literature.

8

Conclusion and Future work

My PhD thesis focuses on design compilation and test support for dataflow applications. Both parts are necessary to go from self-timed circuits to large-scale hardware systems. I started with (1) an existing ARCwelder compiler that mapped dataflow applications to Click circuits, and (2) an existing test approach that worked on GasP circuits. In this thesis I extend the ARCwelder compiler to support a larger class of GasP circuits than existed beforehand. This larger class was necessary to support the already compiler-supported dataflow operations. I also extend the existing test approach to one that supports many forms of testing — slow and at speed. The extended test approach can be used to test as well as to debug and characterize the final hardware system and its parts.

My thesis goal was to develop general compilation and test principles for a multitude of self-timed circuit families, including Click and GasP. Now, at the end of my PhD research and developments in design compilation and test, I can report that we indeed have general design and test principles that work for many – if not all – of the well-known self-timed circuit families, including Click and GasP [46]. Though these design and test principles are not an immediate result of my PhD research, my research served as an eye opener by providing pointers for where

to look for common design and test principles.

This eyeopener came approximately midway in my PhD research. As a result, parts of this PhD thesis use the old design approach, before the ARC's 2015 publication, while other parts use the new approach, advocated in our 2015 publication [46].[1]

The old approach is used in the Telescope GasP (TGasP) designs in Chapter 2 and Appendices A–C, in the ARCwelder compiler study in Chapter 3, in those parts of the hierarchical design for test and debug solution presented in Appendices E–F, and in those parts of the arbiter design study presented in Appendices G–H.

The new approach, also known as "Naturalized Communication and Testing," is used in Chapters 4 and 5, which explain the new design and test principles. The new approach separates communication and data storage, done in *links*, from computation and actions done in *joints*. This link-joint model hides the specifics of the communication protocols and the specifics of the data storage inside the links, thus creating a generic link-joint interface that operates in terms of "got data" (full), "got space" (empty), "copy data" (fill), and "no longer need your data" (drain). These terms match well the notion of dataflow in distributed systems.

The link-joint model gives full freedom to work with multiple circuit families and to use each family's strengths and weaknesses to decide who goes where, when, and why.

Given that large parts of this PhD thesis use the old design approach, I will end

[1] Note that the ARC's 2015 publication by Marly Roncken et al. [46] is a joint publication by all team members of the Asynchronous Research Center (ARC).

this thesis by providing guidelines for future work on how to adapt “old approach” results to the new approach:

- The TGasP modules in Chapter 2 must be partitioned into links and joints. More investigation is needed to decide which of the three circuit design solutions makes the most sense in the new approach.
- The ARCwelder compiler discussed in Chapter 3 must be reformulated in terms of links and joints and a general full-empty protocol. I suspect that doing this with care will solve most of the problems I encountered when I extended the existing ARCwelder compiler code from supporting Click circuits to also supporting GasP and Telescope GasP circuits. In the new approach:
  - Initialization can be isolated in the links.
  - Action control can be isolated in the joints.
  - Combinational loops now go from joint to link to joint, through MrGO.
- The hierarchical design for test and debug results presented in Chapter 6 and Appendices E–F can remain mostly unchanged, except that the circuit modules must be partitioned into links and joints. Each circuit family may need its own specific scan read-write library. It would be a good idea to implement my proposal for initialization at power-up, so future chips can be powered up reliably, without experiencing extended drive fights.

References

[1] Link to ARC, the Asynchronous Research Center at Portland State University. http://arc.cecs.pdx.edu/. [cited at p. 80, 128]

[2] Link to ARC's ASYNC 2015 presentation slides on Naturalized Communication and Testing. Downloadable from the ARC web site. 2015. [cited at p. 62, 79, 80]

[3] Link to Fulcrum Microsystems, now part of Intel Corporation. [cited at p. 1, 32]

[4] Link to Matlab. https://www.mathworks.com/. [cited at p. 147]

[5] Link to Tiempo. http://www.tiempo-ic.com/. [cited at p. 1, 32]

[6] M. Abramovici, M. A. Breuer, and A. D. Friedman. *Digital Systems Testing and Testable Design*. Wiley-IEEE Press, 1990. [cited at p. 80, 116]

[7] P. A. Beerel, G. D. Dimou, and A. M. Lines. Proteus: An ASIC Flow for GHz Asynchronous Designs. *IEEE Design & Test of Computers*, 28(5):36–51, September 2011. [cited at p. 1, 32]

[8] M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Springer Publishing Company, 2013. [cited at p. 80]

[9] T. J. Chaney and C. E. Molnar. Anomalous Behavior of Synchronizer and Arbiter Circuits. *IEEE Transactions on Computers*, 22(4):421 – 422, April 1973. [cited at p. 131]

[10] J. Ebergen, B. Coates, and A. Lee. Long-Distance On-chip Communication Using GasP. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 109–116, 2011. [cited at p. 11, 21]

[11] J. Ebergen, J. Gainsley, and P. Cunningham. Transistor Sizing: How to Control the Speed and Energy Consumption of a Circuit. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 51–61, 2004. [cited at p. 138]

[12] S. J. Hollis. Pulse-based, On-chip Interconnect. Technical Report UCAM-CL-TR-698, University of Cambridge, Computer Laboratory, September 2007. [cited at p. 12]

[13] IEEE Std 1149.1-2001. IEEE Standard Test Access Port and Boundary-Scan Architecture. *The Institute of Electrical and Electronics Engineers (IEEE)*. 2001. [cited at p. 100, 121]

[14] ITRS. International Technology Roadmap for Semiconductors: Design. See also: http://www.itrs2.net/2013-itrs.html. 2013. [cited at p. 32]

[15] N. Jamadagni and J. Ebergen. An Asynchronous Divider Implementation. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 97–104, 2012. [cited at p. 35]

[16] N. P. Jamadagni. *Evaluation of Data-Path Topologies for Self-Timed Conditional Statements*. PhD thesis, Electrical and Computer Engineering, Portland State University, 2015. [cited at p. 35]

[17] N. Kluge and R. Wollowski. Completing the Resynthesis Flow for Balsa Circuits by Focusing on the Data Path - first Experiments. In *Handout Book of Abstracts of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 42–43, 2014. [cited at p. 104]

[18] W. Mallon. Bounded Bundled Data. Technical Report ARC2011-wm09, Asynchronous Research Center, Portland State University. Downloadable from the ARC web site. 2011. [cited at p. 10, 11, 14, 46]

[19] W. Mallon and K. Ully. High-performance Self-timed Processing Networks. Technical Report ARC2011-wm08, Asynchronous Research Center, Portland State University. Downloadable from the ARC web site. 2011. [cited at p. 52, 58]

[20] C. Seitz. Chapter 7: System Timing In *C. Mead and L. Conway, Introduction to VLSI Systems* pages 218–262. Addison-Wesley, 1980. [cited at p. 86, 87, 132]

[21] J. T. Mechler. Clock Domain Challenges on Networking Chips: A Design Center Perspective. Keynote at the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), 2013. [cited at p. 1]

[22] S. Mettala Gilla. Library Characterization and Static Timing Analysis of Single-Track Circuits in GasP. Master's thesis, Electrical and Computer Engineering, Portland State University, 2010. [cited at p. 48]

[23] S. Mettala Gilla, M. Roncken, and I. Sutherland. Long-Range GasP with Charge Relaxation. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 185–195, 2010. [cited at p. 12, 15]

[24] S. Mettala Gilla, M. Roncken, and I. Sutherland. Verilog Simulations for GasP Circuits Designed in Electric. Technical Report ARC2012-smg06, Asynchronous Research Center, Portland State University, 2012. [cited at p. 51]

[25] S. Mettala Gilla, M. Roncken, and I. Sutherland. GasP Storage and Telescope GasP Broadcast: Store, Amplify, Fork and Join. Technical Report ARC2012-smg02, Asynchronous Research Center, Portland State University, 2012. [cited at p. 173]

[26] S. Mettala Gilla, M. Roncken, and I. Sutherland. Telescope GasP Narrowcast and Arbitrated or Data Driven Control: Merge, Branch, and Distribute. Technical Report ARC2012-smg03, Asynchronous Research Center, Portland State University, 2012. [cited at p. 206]

[27] S. Mettala Gilla, M. Roncken, and I. Sutherland. Telescope GasP: Repeat. Technical Report ARC2012-smg04, Asynchronous Research Center, Portland State University, 2012. [cited at p. 262]

[28] S. Mettala Gilla, M. Roncken, and I. Sutherland. Arbiter Design Improvements for GasP. Technical Report ARC2012-smg05, Asynchronous Research Center, Portland State University, 2012. [cited at p. 350]

[29] S. Mettala Gilla, M. Roncken, and I. Sutherland. A GasP versus Click Throughput Comparison: Fibonacci. Technical Report ARC2013-smg03, Asynchronous Research Center, Portland State University, 2013. [cited at p. 52, 58]

[30] S. Mettala Gilla, M. Roncken, and I. Sutherland. A GasP versus Click Throughput Comparison: GCD. Technical Report ARC2013-smg04, Asyn-

chronous Research Center, Portland State University, 2013. [cited at p. 52, 55, 58]

[31] S. Mettala Gilla, M. Roncken, and I. Sutherland. A GasP versus Click Throughput and Latency Comparison: LEETL. Technical Report ARC2013-smg05, Asynchronous Research Center, Portland State University, 2013. [cited at p. 52, 55, 58]

[32] S. Mettala Gilla, M. Roncken, and I. Sutherland. User Guide for Click and GasP Design Experiments in ARCwelder. Technical Report ARC2013-smg02, Asynchronous Research Center, Portland State University, 2013. [cited at p. 58]

[33] S. Mettala Gilla, M. Roncken, and I. Sutherland. GasP Scan Chain Implementation for Init, Go, and Single Step. Technical Report ARC2013-smg08, Asynchronous Research Center, Portland State University, 2013. [cited at p. 290]

[34] S. Mettala Gilla, M. Roncken, and I. Sutherland. Scan Testing GasP using a JTAG Controller. Technical Report ARC2013-smg09, Asynchronous Research Center, Portland State University, 2013. [cited at p. 308]

[35] S. Mettala Gilla, M. Roncken, I. Sutherland, and N. Jamadagni. Verilog: Models in Electric for JTAG, Scan, and GasP Designs. Technical Report ARC2014-smg03, Asynchronous Research Center, Portland State University, 2014. [cited at p. 34, 51]

[36] S. Mettala Gilla, M. Roncken, and I. Sutherland. Further Arbiter Design Improvements: Noise Robustness. Technical Report ARC2014-smg01, Asynchronous Research Center, Portland State University, 2014. [cited at p. 372]

[37] C. E. Molnar, I. W. Jones, W. S. Coates, and J. K. Lexau. A FIFO Ring Performance Experiment. In *Proceedings of the IEEE International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 279–289, 1997. [cited at p. 85, 87, 103]

[38] L. W. Nagel and D. O. Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). In *Proceedings of the 16th Midwest Symposium on Circuit Theory*, 1973. [cited at p. 149]

[39] H. Park. *Formal Modeling and Verification of Delay-Insensitive Circuits*. PhD thesis, Electrical and Computer Engineering, Portland State University, 2015. [cited at p. 34, 36, 46, 49, 61]

[40] H. Park, A. He, S. Mettala Gilla, and M. Roncken. Cheatsheet for ARCwelder. Technical Report ARC2013-hp01, Asynchronous Research Center, Portland State University, 2013. [cited at p. 34]

[41] H. Park, A. He, M. Roncken, X. Song, and I. Sutherland. Modular Timing Constraints for Delay-Insensitive Systems. *Journal of Computer Science and Technology (JCST)*, 31(1):77–106, January 2016. [cited at p. 36, 46, 48, 49, 61]

[42] A. Peeters and A. Bink. Asynchronous Circuit Technology Is On The Market (Keynote). In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), page xiv*. Downloadable from the ASYNC 2008 web site. 2008 [cited at p. 3, 33]

[43] A. Peeters, F. te Beest, M. de Wit, and W. Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems*

(ASYNC), pages 3–14, 2010. [cited at p. 2, 3, 10, 11, 19, 33, 37, 42, 65, 76]

[44] M. Roncken. Defect-Oriented Testability for Asynchronous ICs. *Proceedings of the IEEE*, 87(2):363–375, February 1999. [cited at p. 87]

[45] M. Roncken, C. Cowan, B. Massey, S. Mettala Gilla, H. Park, R. Daasch, A. He, Y. Hei, J. W. Hunt Jr., X. Song, and I. Sutherland. Beyond Carrying Coal To Newcastle: Dual Citizen Circuits. A. Mokhov (Ed.): This Asynchronous World - Essays dedicated to Alex Yakovlev on the occasion of his 60th birthday, Pages 241–261, Newcastle University, UK, 2016. [cited at p. 6]

[46] M. Roncken, S. Mettala Gilla, H. Park, N. Jamadagni, C. Cowan, and I. Sutherland. Naturalized Communication and Testing. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 77–84, 2015. [cited at p. 2, 5, 7, 8, 51, 59, 60, 62, 78, 80, 86, 88, 103, 124, 161, 162, 281]

[47] M. Roncken and I. Sutherland. Initialize, Go and Single Step. Technical Report ARC2013-is04, Asynchronous Research Center, Portland State University, 2013. [cited at p. 127]

[48] S. M. Rubin. Using the ELECTRIC^{TM} VLSI Design System, Version 8.11. *Static Free Software and Sun Microsystems*, ISBN 0-9727514-3-2, R.L. Ranch Press, 2010. [cited at p. 45]

[49] M. Singh and S. M. Nowick. MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(6):684–698, June 2007. [cited at p. 65]

[50] J. Sparsø and S. Furber (Eds.). *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001. [cited at p. 6, 8, 11, 132]

[51] I. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989. [cited at p. 65]

[52] I. Sutherland. Fourth Class Handout – Proper Stopper. Technical Report ARC2010-is49, Asynchronous Research Center, Portland State University, 2010. [cited at p. 103, 130]

[53] I. Sutherland. GasP Circuits that Work. *ECE507 Course, Fall 2010, Asynchronous Research Center, Portland State University*, See: http://arc.cecs.pdx.edu/fall10, 2010. [cited at p. 37, 42, 65, 130]

[54] I. Sutherland. The Tyranny of the Clock. *Communications of the ACM*, 55(10):35–36, October 2012. [cited at p. 1]

[55] I. Sutherland and S. Fairbanks. GasP: A Minimal FIFO Control. In *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, ASYNC, pages 46–53, 2001. [cited at p. 2, 15, 37, 65]

[56] I. Sutherland, M. Roncken, N. Jamadagni, C. Cowan, and S. Mettala Gilla. The Weaver, an 8x8 Crossbar Experiment. Technical Report ARC2015-is07v11, Asynchronous Research Center, Portland State University. Downloadable from the ARC web site. 2015. [cited at p. 51, 88, 128]

[57] I. Sutherland, M. Roncken, and S. Mettala Gilla. The Master Clear Blunder. Technical Report ARC2013-is21, Asynchronous Research Center, Portland State University, 2013. [cited at p. 127]

[58] I. Sutherland, B. Sproull, and D. Harris. *Logical Effort: Designing Fast CMOS Circuits*. Morgan Kaufmann Publishers, 1999. [cited at p. 138, 156]

[59] F. te Beest, A. Peeters, M. Verra, K. van Berkel, and H. Kerkhoff. Automatic Scan Insertion and Test Generation for Asynchronous Circuits. In *Proceedings of the IEEE International Test Conference (ITC)*, pages 804–813, 2002. [cited at p. 2]

[60] K. van Berkel, R. Burgess, J. Kessels, M. Roncken, F. Schalij, and A. Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design and Test of Computers*, 11(2):22–32, August 1994. [cited at p. 11, 33, 104]

[61] N. H. E. Weste and D. Money Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, 2011. [cited at p. 80]

[62] S. Zeidler and M. Krstić. A Survey about Testing Asynchronous Circuits. In *IEEE Proceedings of the European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4, 2015. [cited at p. 1]

Appendix A

Telescope GasP: Storage and Broadcast

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapters 2 and 3 and can be found as reference [25] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland.
  GasP Storage and Telescope GasP Broadcast: Store, Amplify, Fork and Join. *Technical Report, ARC2012-smg02, Asynchronous Research Center, Portland State University*, March, 2012.

Asynchronous Research Center

Portland State University

Subject: **GasP Storage and Telescope GasP Broadcast: Store, Amplify, Fork and Join**
Date: **1 March, 2012**
From: **Swetha Mettala Gilla, Marly Roncken and Ivan Sutherland**
ARC#: **2012-smg02**

References:

[1] Peter Beerel, Georgios Dimou, and Andrew Lines. Proteus: an ASIC Flow for GHz Asynchronous Designs, *IEEE Design & Test of Computers, Vol. 28, Issue 5*, pp. 36-51, 2011.

[2] Andrew Bardsley, Luis Tarazona, and Doug Edwards. Teak: A Token-Flow Implementation for the Balsa Language, In *Proceedings of the International Conference on the Application of Concurrency to System Design (ACSD)*, pp. 23-31, 2009.

[3] Teak asynchronous synthesis, Univ. of Manchester: http://apt.cs.man.ac.uk/projects/teak/.

[4] Jo Ebergen, Bill Coates, and Austin Lee. Long-Distance On-Chip Communication using GasP. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 119-116, 2011.

[5] Willem Mallon and Ivan Sutherland. Icons for Click and GasP Modules, *ARC2011-is05, Technical Report, Asynchronous Research Center, Portland State University,* March 2011.

[6] Asynchronous Research Center website: http://arc.cecs.pdx.edu/.

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

[8] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Telescope GasP: Overview. *ARC2012-smg01, Technical Report, Asynchronous Research Center, Portland State University,* 2012. **Note: ARC2012-smg01 has been fully integrated into Chapter 2 of Swetha's Ph.D. thesis.**

[9] — GasP Storage and Telescope GasP Broadcast: Store, Amplify, Fork, and Join. *ARC2012-smg02.* **Note: ARC2012-smg02 is included as Appendix A in Swetha's Ph.D. thesis.**

[10] — Telescope GasP Narrowcast and Arbitrated or Data Driven Control: Merge, Branch, and Distribute. *ARC2012-smg03.* **Note: ARC2012-smg03 is included as Appendix B in Swetha's Ph.D. thesis.**

[11] — Telescope GasP: Repeat. *ARC2012-smg04.* **Note: ARC2012-smg04 is included as Appendix C in Swetha's Ph.D. thesis.**

[12] — Arbiter Design Improvements for GasP. *Technical Report, ARC2012-smg05, Work in Progress.* **Note: ARC2012-smg05 is now finished and included as Appendix G in Swetha's Ph.D. thesis.**

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

[14] Steven M. Rubin. Using the ELECTRIC™ VLSI Design System, Version 8.11. *Static Free Software and Sun Microsystems, ISBN 0-9727514-3-2,* R.L. Ranch Press, 2010.

[15] Charles Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems, Carver Mead and Lynn Conway (Eds), Addison-Wesley*, pages 218-262, 1980.

[16] Ivan Sutherland. A Mutual Exclusion Pass Gate, *ARC2010-is26, Technical Report, Asynchronous Research Center, Portland State University,* May 2010.

[17] Ivan Sutherland. Fourth Class Handout: Proper Stopper, *ARC2012-is49, Technical Report Asynchronous Research Center, Portland State University,* October 2010.

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

[19] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow, In *Proc. International Workshop on Logic Synthesis, 2004.*

This document contains information developed at the Asynchronous Research Center at Portland State University. Disclose this information to whomever you choose. Distribution is permitted with adequate reference to the source of the ideas and information. You may reproduce this material for any educational use. Copies of the material must contain this notice.

Abstract

This document gives Telescope GasP implementations for pipeline modules for storage and broadcast communication used in ARCwelder [5,6]. ARCwelder, previously called TPDesign, is a design and compilation environment for dataflow computations; we use it to design self-timed systems. The organization of ARCwelder builds upon results and key learnings of the data-driven compiler effort by Handshake Solutions [13]. Related self-timed compilation efforts can be found in for instance [1,2,3,19].

An overall motivation for Telescope GasP, including pros, cons, and alternatives, can be found in [8]. Telescope GasP implementations for narrowcast 1-to-1ofN or 1ofN-to-1 communications, or 1-to-1toNofN communications follow in [10]. Telescope GasP implementations for Repeat modules follow in [11]. The references in the current ARC report list all references on Telescope GasP used here or in [8,9,10,11].

Keywords: GasP backend for ARCwelder (TPDesign), Telescope GasP for storage and broadcasting.

Notation:

1. All GasP implementations in this document and in the related documents [8,9,10,11] are non-inverting, i.e. their handshake channels, a.k.a. statewires, use the same encoding. Statewires say whether the data bundled with the statewire are valid. We use:
   - HI for a high voltage level, e.g. VDD, to indicate a handshake REQUEST phase with valid data.
   - LO for a low voltage level, e.g. VSS or GND, to indicate a handshake ACKNOWLEDGE phase where data are no longer needed.

   In our circuit diagrams, we use a short 45-degree line segment for VDD and a triangle for VSS. A PMOS transistor connected to VDD and an NMOS transistor connected to VSS are depicted as:

2. Our schematic designs use a so-called JBOX construct. This is a special construct in Electric [14] to connect signals with different names. The symbol for a JBOX construct is a box with the letter J in it. We specifically use it to join the **master clear (mc)** signals on the various statewires of a module. For example, **Figure 8** on page 12 uses a JBOX to connect master clear signals in[mc] and out[mc].

3. It turns out that the telescoping handshake relation requires more relative timing constraints than are required for conventional GasP [18,7]. The constraints that we need to obtain a telescopic handshake relation tend to be transition specific. If the delay margins are small, e.g. in the order of 1 gate delay, we may want to add extra delay circuitry to increase the margin from say 1 to 3 gate delays. To delay a transition from **sin** LO to **sout** HI by two additional gate delays we will use the left-hand circuit in **Figure 1** below, and to delay a transition from **sin** HI to **sout** LO by two additional gate delays, we will use the right-hand circuit in **Figure 1**.

   The two delay circuits in **Figure 1** add two gate delays to the targeted **sin-**to-**sout** transition. Depending on the actual cycle time of the module in the given data path, we can extend this further to four additional gate delays or more, without losing throughput. These two types of delay circuits are sufficiently generic for our needs, and have the following properties:
   - The required transition is slowed down by sufficient margin.
   - The required transition is fast enough to sustain the maximum throughput.
   - The delay of the opposite transition matches the original delay without the delay circuit.

   When we present the circuit schematics of a telescope GasP component, we will indicate places with a 1 gate delay margin where one of the delay circuits in **Figure 1** may be needed.

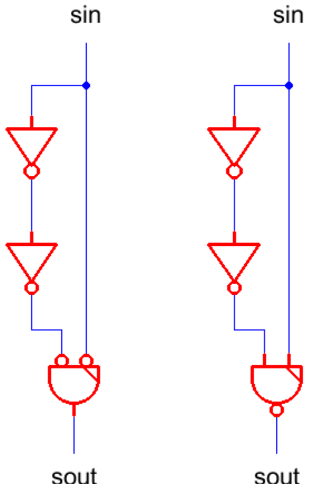


Figure 1: Delay circuits to increase the delay margin from **sin** LO to **sout** HI by two gate delays (left), and to increase the margin from **sin** HI to **sout** LO by two gate delays (right). Other path delays remain unchanged.

Acknowledgements: We gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD thesis work.

Table of Contents

References 1
1. INTRODUCTION 4
2. Custom Driver Designs for Storage & Broadcast Modules 6
   2.1. Single-Track Statewire Drivers for Pull-HI and Keep-LO 6
   2.2. Single-Track Statewire Drivers for Pull-LO and Keep-HI 7
3. Store Module 9
4. Amplify Module 11
5. Fork Modules 17
   5.1. 2-Way Fork Module, or Fork2 17
   5.2. 3-Way Fork Module, or Fork3 23
6. Join Modules 24
   6.1. 2-Way Join Module, or Join2 24
   6.2. 3-Way Join Module, or Join3 30
7. Conclusion and Future Work 32

1. INTRODUCTION

This document gives GasP control implementations for the following pipeline modules used in ARCwelder and specified and listed in [5,6]: Store, Amplify, Fork, and Join. These modules facilitate storage and broadcasting. Below follows a short description of each module. We will leave out handshake details and focus on dataflow aspects instead. We grouped some of the modules to emphasize their similarity.

- **Store**
  - Our GasP implementation for the Store module is a 6-4 GasP basic FIFO module. It is the only module in this document that clocks latches in the data path to store data.
  - The design details of the Store module follow in Section 3 of this document.
- **Amplify**
  - The Amplify module is the self-timed version of the well-known “repeater” module in synchronous designs. It is used in the control logic to bridge long module-to-module distances with minimal latency. Our two Telescope GasP implementation versions are based on [4] but are non-inverting. In a sense, they form a Telescope GasP version of the basic FIFO module without storage.
  - The two implementations follow in Section 4 of this document.
- **Fork and Join**
  - Fork and Join modules form a companion pair for broadcast communication, providing one input to N-way outputs (Fork) and N-way inputs to one output (Join). We give Telescope GasP implementations for two-way and three-way Fork and Join modules.
  - Implementation details for the Fork modules follow in Section 5 of this document.
  - Implementation details for the Join modules follow in Section 6 of this document.

When designing these modules, we distinguish and design four communication control actions:

1. **Forward Transfer**
   The HI handshake request signal coming in from the predecessor statewire transfers into a HI request signal on the successor statewire. The actual transfer generally requires synchronization with other incoming and with outgoing signals. We opted for a forward transfer latency of 6 gate delays for the Store module and of 2 gate delays for the other modules.
2. **Backward Transfer**
   The LO handshake acknowledge signal coming in from the successor statewire transfers into a LO acknowledge signal on the predecessor statewire. The transfer generally requires synchronization with other outgoing and with incoming signals. We opted for a backward transfer latency of 4 gate delays for the Store module and of 2 gate delays for the other modules.
3. **Forward Reset**
   The forwarded HI request signal on the successor statewire turns off its own HI drive after 5 gate delays, thereafter relying on the keeper to keep the statewire HI as long as needed. The Forward Reset in a storage-free Telescope GasP module strictly precedes its Backward Reset. Traditional GasP modules, like module Store, turn off their forward HI and backward LO drives concurrently.
4. **Backward Reset**
   In a storage-free Telescope GasP module, the backward transferred LO acknowledge signal on the predecessor statewire turns off its own LO drive after 5 gate delays. Traditional GasP modules, like module Store, turn off their forward HI and backward LO drives concurrently.

The two transfer parts each contain a driver and a half-keeper. Similar to the half-keeper design organizations of 4-2 Gasp in [18] and of 6-4 GasP in [7], the half-keeper serves two purposes: it prevents the statewire from floating when the drive ceases, and it avoids a keeper-driver short-current conflict.

We will explore two different approaches in designing Telescope GasP modules. The first approach uses separate self-resetting loops for incoming and outgoing handshake channels. The Amplify module in **Figure 8** is an example of this first design approach. It has two self-resetting loops: the loop at the top is used to start and stop the HI drive for *out[sw],* while the loop at the bottom is used to start and stop the LO drive for *in[sw].* The second approach shares self-resetting loops between incoming and outgoing

handshake channels where possible. The Amplify module in the following **Figure 9** is an example of the latter design approach. It has one self-resetting loop, with separate taps to start and stop the HI respectively LO drives for *out[sw]* and *in[sw].* The second design approach is based on the amplifier design in [4], but we generalized the approach to other modules beyond Amplify.

Telescope GasP implementations designed according to the first approach are saved under library name *TelescopeGasP_SeparateStates*. Implementations designed according to the second approach are saved under *TelescopeGasP_wStateSharing*.

Both approaches have their advantages and disadvantages. The advantage of the first approach used in **Figure 8** is that its control-data bundling constraints are less strict than those for the second approach used in **Figure 9**. This is not immediately obvious from the Amplify design, but it will become obvious when we get to the data-controlled Distribute modules. For details, see [10].

The advantage of the second approach used in **Figure 9** is that it has larger delay margins than the first approach. The second approach has delay margins that we feel are safe, whereas the first approach has marginal relative timings that we may want to adjust. **Figure 8** has two such marginal relative timings:

1. After *in[sw]* rises, it takes 2 gate delays before *out[sw]* rises and disables the inverted input AND gate in the lower self-resetting loop, while it takes 3 gate delays for the HI *in[sw]* signal to enable that same AND gate via the lower self-resetting loop. To guarantee a telescope relation between *in[sw]* and *out[sw],* the inverted input AND gate must sense the disabling *out[sw]* transition before it senses the enabling *in[sw]* transition. The delay margin between the two is only 1 gate delay.
2. After *out[sw]* falls, it takes 2 gate delays before *in[sw]* falls and disables the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the LO *out[sw]* signal to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *in[sw]* and *out[sw],* the NAND gate must sense the disabling *in[sw]* before it senses the enabling *out[sw]* transition. The delay margin between the two is only1 gate delay.

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can make both margins as safe as the delay margins deployed in the second design approach, by adding extra circuitry to delay the transition that must arrive later by 2 more gate delays. Other transitions remain unaffected. For instance, to delay a transition from **sin** LO to **sout** HI we insert the left-hand circuit of **Figure 1**, and to delay a transition from **sin** HI to **sout** LO we insert the right-hand circuit of **Figure 1**.

One of the design criteria for the current Telescope GasP implementations is a small 2 gate delay latency between incoming and outgoing handshakes. From [8], we know that we need *at least* 2 gate delays to implement non-inverting Telescope GasP. Our design criterion is to make this latency *no more than* 2 gate delays. To be more precise: we want a minimum latency of 2 gate delays per module from the moment its last parallel incoming handshake goes HI (starts) to the moment its outgoing handshakes go HI (start), and, likewise, a minimum latency of 2 gate delays from the moment its last outgoing handshake goes LO (resets) to the moment its parallel incoming handshakes go LO (reset).

To guarantee this low latency, some of the control functionality is inevitably forced into the drivers for the handshake channels. This results in a myriad of custom driver designs, each with 1 gate delay latency. Section 2 below lists all custom design drivers used for the design of Store, Amplify, Fork, and Join, and includes their circuit schematics.

2. Custom Driver Designs for Storage & Broadcast Modules

To achieve a small 2 gate delay transfer latency, some of the forward and backward transfer logic is inevitably integrated into the driver and half-keeper logic. As a result, Telescope GasP modules use a large variation of driver and half-keeper logic for incoming and outgoing statewires - larger than we are used to see in traditional, e.g. 6-4, GasP modules.

Section 2.1 gives the implementations of HI driver and LO half-keeper logic for successor drivers of outgoing statewires of storage and broadcast modules. Section 2.2 does the same for the LO driver and HI half-keeper logic for predecessor drivers of incoming statewires. By convention, the initial value of a statewire is LO, and so the **master clear (mc)** signal and related circuitry for fast return-to-zero resetting appear in the predecessor drivers.

2.1. Single-Track Statewire Drivers for Pull-HI and Keep-LO

Our implementations of Store, Amplify, Fork, and Join use the following HI driver and LO half-keeper logic for successor drivers:

- **Succ Driver:** A LO input drives the statewire HI, and a half-keeper keeps the statewire LO.
- **Succ AND Driver:** The AND of two LO inputs drives the statewire HI, and a half-keeper keeps it LO.

Both circuits have an input-to-output transition time of one gate delay. Their implementations follow in **Figure 2** and **Figure 3** below.

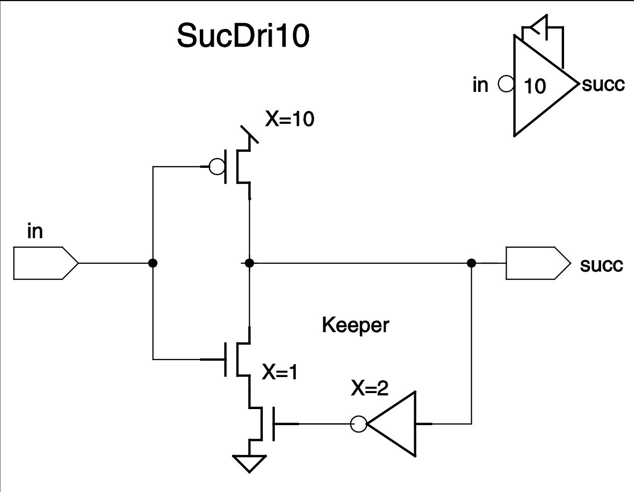


Figure 2: Transistor-level GasP design for **Succ Driver**, with one input to drive the statewire HI within one gate delay, and a half-keeper to keep it LO. Its input signal is named **in**. Its statewire output is named **succ**. The official name of this gate in the 180nm library kept in Electric [14] is **SucDri10**. We will use the icon in the top right corner of the picture as a gate-level shortcut for the design. SucDri10 is used in the 6-4 GasP design of the Store module, and in the Telescope GasP design of the Amplify and Fork modules, and in the first design implementation for the two-way Join.

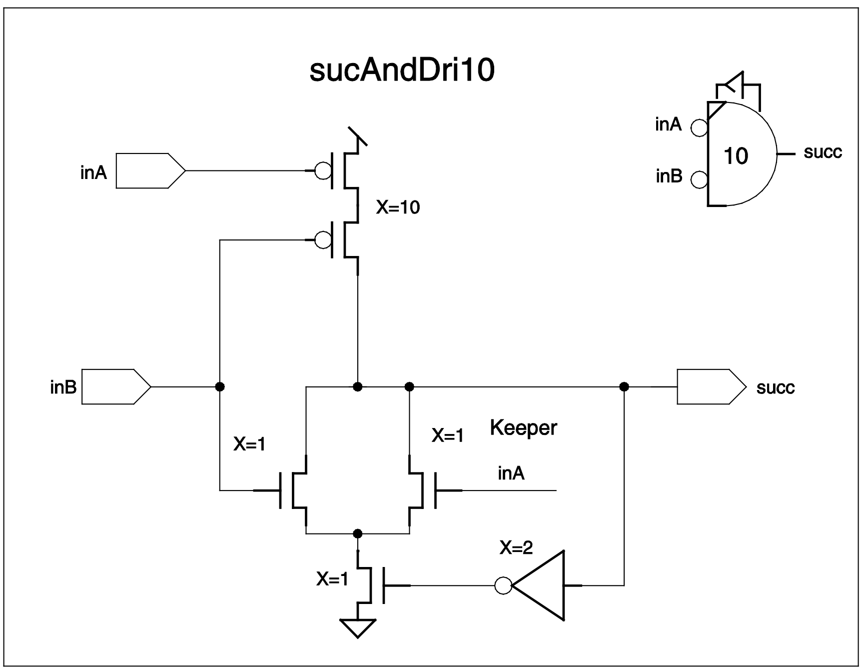


Figure 3: Transistor-level GasP design for **Succ AND Driver**, with library cell name **SucAndDri10**. When inputs **inA** and **inB** are LO, statewire **succ** is driven HI within one gate delay. The half-keeper keeps **succ** LO, when needed. The top-right icon serves as gate-level shortcut. SucAndDri10 is used in the second design implementation for the two-way Join and in the three-way Join.

2.2. Single-Track Statewire Drivers for Pull-LO and Keep-HI

Our implementations of Store, Amplify, Fork, and Join use the following LO driver and HI half-keeper logic for predecessor drivers:

- **Pred Driver:** A HI input drives the statewire LO, and a half-keeper keeps it HI.
- **Pred AND Driver:** The AND of two HI inputs drives the statewire LO and a half-keeper keeps it HI.

Both circuits have an input-to-output transition time of one gate delay. Their implementations follow below in **Figure 4** and **Figure 5**. By default, statewires are initialized LO. So, unless otherwise specified, LO drivers have a **master clear (mc)** input signal and related circuitry to initialize the statewire to LO. All driver implementations in this Section follow the default initialization settings. That means that also any additional circuitry for fast (return-to-zero) resetting goes into the LO driver logic for incoming statewires.

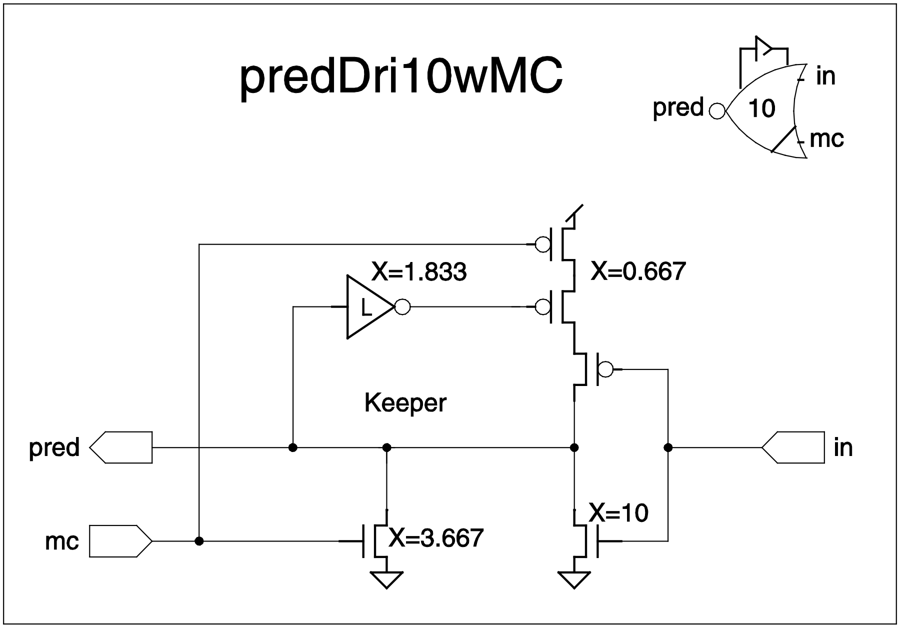


Figure 4: Transistor-level GasP design for **Pred Driver**, with one input **in** to drive statewire **pred** LO within one gate delay, and a half-keeper to keep pred HI. The official cell name is **predDri10wMC**. In addition to **in**, there is an extra input signal **mc**, for **master clear**, which we use to initialize **pred** to LO. We will use the icon in the top right corner of the picture as a gate-level shortcut for this design. It is used in the 6-4 GasP design of the Store module, and in the Telescope GasP design of the Amplify module, in the first design implementation of the two-way Fork, and in the Join modules.

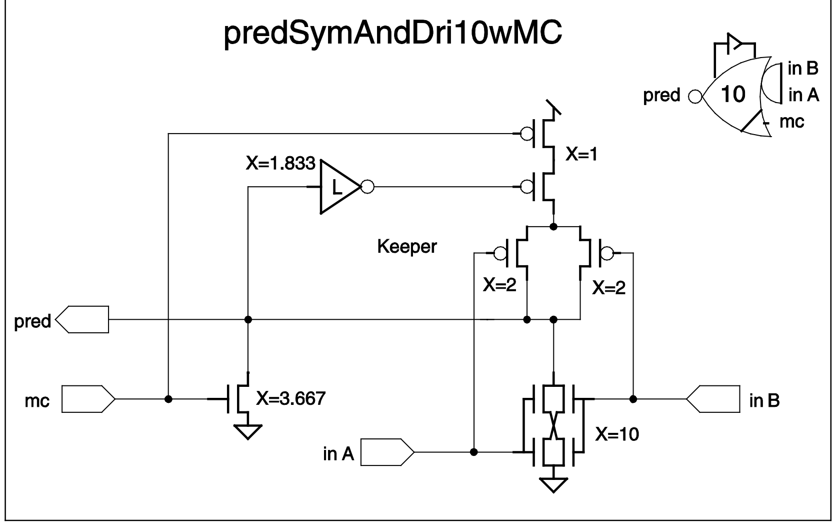


Figure 5: Transistor-level GasP design for **Pred AND Driver**. Its official name in the 180nm cell library kept in Electric [14] is **predSymAndDri10wMC**. When both inputs **inA** and **inB** are high or input signal **mc** is high, statewire **pred** is driven LO within one gate delay. The symmetric NMOS AND pair in the design makes the gate delay independent of the arrival ordering of the inputs. The half-keeper is there to keep **pred** HI, when needed. Global input signal **mc**, for **master clear**, is used to initialize **pred** to LO. We will use the icon in the top right corner as a gate-level shortcut for this design. It is used in the second design implementation of the two-way Fork, and in the three-way Fork module.

3. Store Module

Data storage is indicated explicitly in ARCwelder [5,6]. For the time being, we'll separate storage and computation. We anticipate that for future low-latency designs, we may combine more complex control features into storage elements.

The basic Store control module is implemented as a 6-4 GasP FIFO module. Its circuit diagram follows in **Figure 6**. The Forward Transfer part uses the Succ Driver of **Figure 2**. There are 5 additional gates between input **in[sw]** of the Store module and the input of the Succ Driver, giving a total forward latency of 6 gate delays for the Store module.

The Backward Transfer part is implemented by the Pred Driver of **Figure 4**. There are 3 additional gates between output **out[sw]** of the Store module and the input of the Pred Driver, giving a total backward latency of 4 gate delays for the Store module.

In addition to a master clear signal **mc,** to initialize the module, and input and output statewires **in[sw]** and **out[sw]**, the module has an output signal, called **cl**. Signal **cl** acts as a local or source-synchronous clock signal. It controls latches or flip-flops in the data path, to make these store new or keep old data.

The behavior of the Store module can be described as follows. Initially, **in[sw]**, **out[sw]** and **mc** are LO. When **in[sw]** goes HI, indicating the presence of data, and **out[sw]** is LO, indicating the availability of space, clock signal **cl** goes HI. Signal cl going HI results in the following three **parallel** sets of actions:

1. Latches become transparent and flip-flops receive new data in their master latches.
2. The Forward and Backward Transfer sections report the presence of new data to the successor and new availability of space to the predecessor modules by driving **out[sw]** HI and **in[sw]** LO.
3. The Forward and Backward Reset sections kick in, and cut off the HI clock pulse, the HI forward drive, and the LO backward drive to a 5 gate delay pulse signal each. At the end of the HI clock pulse, the latches become opaque and the flip-flops copy the new data to their slave latches and to their outputs. Data outputs of the latches and flip-flops now contain stable and new data.

At the end of these three **parallel** sets of actions, the Store module waits until **out[sw]** is LO and **in[sw]** is HI again, so it can start its next cycle.

The minimum cycle time is 10 gate delays, as set by the 5 gate delay loops in the circuit diagram of **Figure 6**. The right-hand loop executes the Forward Transfer and Forward Reset sections. The left-hand loop executes the Backward Transfer and Backward Reset sections. Both loops start simultaneously with the negated-input AND gate in the middle. The loops share 3 gates: the negated-input AND, and its two subsequent inverters or rather inverting amplifiers.

In **Figure 7** we give a collection of SPICE-level simulation waveforms of our 6-4 GasP implementation of the Store module. The waveforms were programmed and viewed from the Electric design environment [14] that we use for designing GasP modules. The waveforms support the above behavioral description.

Note

Figure 6 uses a JBOX to connect in[mc] to out[mc]. As also explained in Introduction Section 1, a JBOX is special construct in Electric [14] to connect, i.e. join, signals with different names. The symbol for a JBOX construct is a box with the letter J in it. We specifically use it to join the **master clear (mc)** signals on the various statewires of a module, as we do here for in[mc] and out[mc].

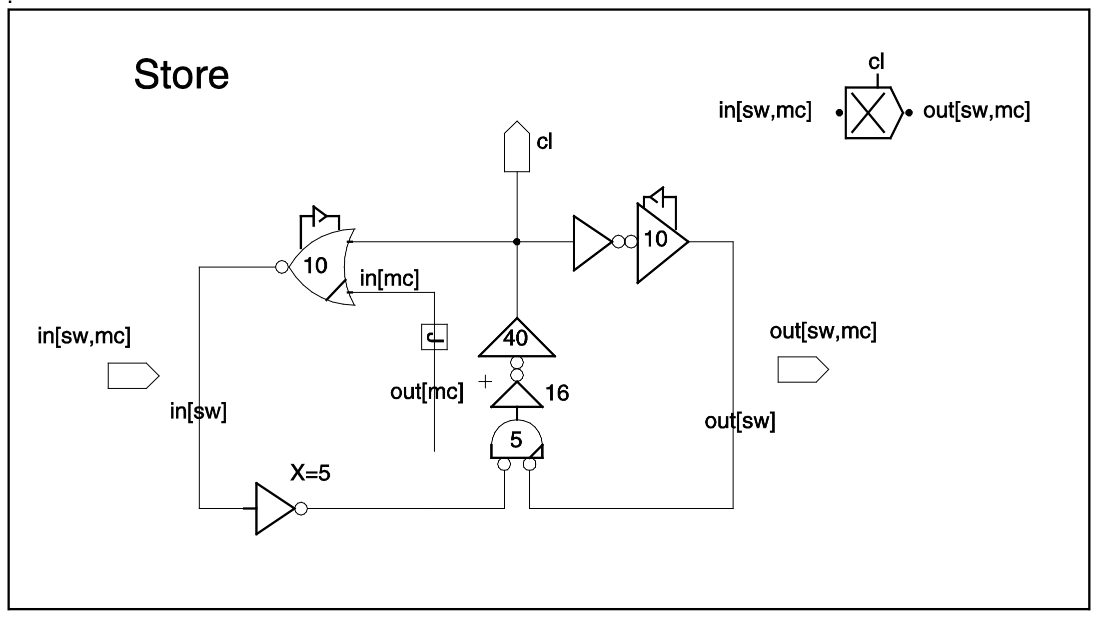


Figure 6: Circuit for 6-4 GasP control module Store, with input statewire and master clear **in[sw,mc],** output statewire and joined master clear **out[sw,mc]**, and local clock signal **cl** to clock and store data in the bundled data path. We will use the icon in the top-right corner to represent this GasP Store module.

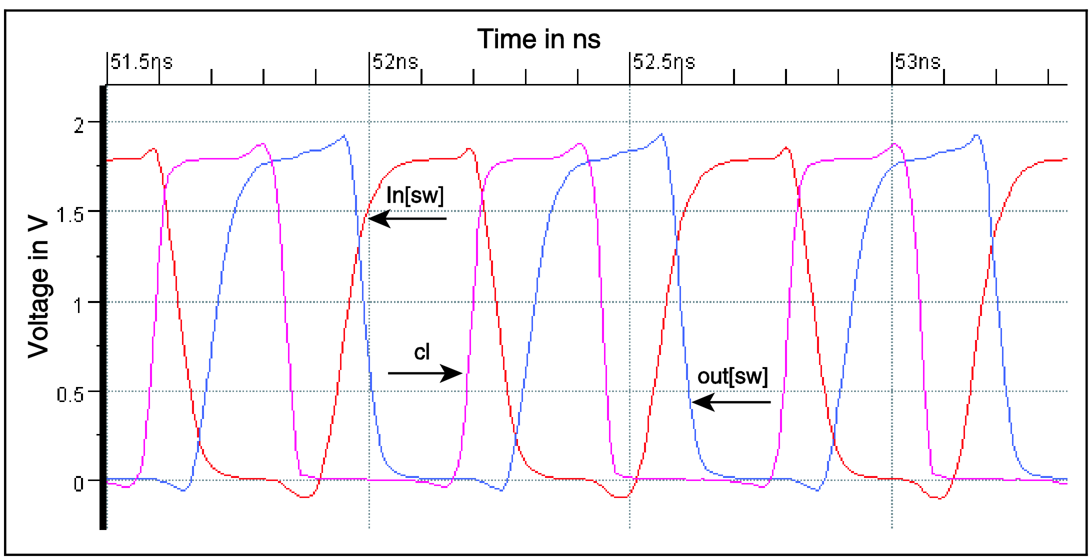


Figure 7: SPICE-level simulation of the 6-4 GasP implementation of the Store module in **Figure 6**. Red signal in[sw] and blue signal out[sw] swap their states in parallel and create a HI pulse on the clock signal, whenever in[sw] is HI (FULL) and out[sw] is LO (EMPTY). After in[sw] has gone LO, the predecessor in the simulation test environment switches in[sw] back to HI. Likewise, after out[sw] has gone HI, the successor in the simulation test environment switches out[sw] back to LO. Then, the cycle starts again.

4. Amplify Module

The Amplify module is the self-timed version of the well-known "repeater" module in synchronous designs. It is used in the control logic to bridge long module-to-module distances with minimal latency.

The implementation of the Amplify module in **Figure 8** is an example of our first design approach with independent self-resetting loops for statewires of incoming and outgoing handshake channels. It has two self-resetting loops: the loop at the top is used to start and stop the HI drive for *out[sw],* while the loop at the bottom is used to start and stop the LO drive for *in[sw].*

The Amplify module implementation in **Figure 9** is an example of the second design approach with shared self-resetting loops for the statewires of incoming and outgoing handshake channels. It has one self-resetting loop, with separate taps to start and stop the HI respectively LO drive for *out[sw]* and *in[sw].* **Figure 9** is based on the amplifier design in [4], but uses non-inverting logic between incoming and outgoing handshakes.

SPICE simulations of the handshake behavior of the Amplify implementations in **Figure 8** and **Figure 9** follow in **Figure 10** and **Figure 11**(top). The handshake behaviors are the same, and can be described as follows. Initially, **in[sw]**, **out[sw]** and **mc** are LO. When **in[sw]** goes HI, indicating the presence of data, and **out[sw]** is LO, indicating the availability of space, the following four sets of actions are started **in sequence,** ordered as indicated:

1. The Forward Transfer part signals new data to the successor module by driving **out[sw]** HI. It takes 2 gate delays to go from **in[sw]** HI to **out[sw]** HI.
2. The Forward Reset part kicks in, cutting off the HI forward drive to a 5 gate delay pulse signal.
3. The Backward Transfer part waits until **out[sw]** is LO, and then reports new availability of space to the predecessor module by driving **in[sw]** LO, 2 gate delays after receiving **out[sw]** LO.
4. The Backward Reset part kicks in, cutting off the LO backward drive to a 5 gate delay pulse.

At the end of these four **serial** sets of actions, the Amplify module waits until **in[sw]** is HI again, so it can start its next cycle.

Internally, the behaviors of the two Amplify implementations differ. The advantage of the implementation in **Figure 9** is that it has delay margins that are relatively safe. In contrast, the implementation in **Figure 8** has two marginal delay margins that we may want to increase to guarantee that the **in[sw]-out[sw]** handshakes really telescope. The two marginal delay margins in **Figure 8** are as follows:

1. The path to disable the negative-input AND gate, which starts with in[sw] HI and then goes the NAND gate and Succ Driver to out[sw] HI, must have less delay than the path to enable the gate from the same starting point, in[sw] HI, through the lower loop. The delay margin between these two paths is only 1 gate delay.
2. Likewise, the path to disable the NAND gate, which starts with out[sw] LO and then goes through the negative-input AND gate and Pred Driver to in[sw] LO, must have less delay than the path to enable the gate from the same starting point, out[sw] LO, through the upper loop. The delay margin between these two paths is only 1 gate delay.

We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Amplify module in **Figure 8** as safe as those in **Figure 9.** Details on the corresponding safety margins for the implementation of the Amplify module in **Figure 9** follow in **Figure 11(middle and bottom)** – see caption for details.

The two implementations have the same cycle time and latencies. In both implementations, the Forward Transfer part is implemented using the Succ Driver of **Figure 2**. There is 1 additional gate between incoming handshake channel **in[sw]** and the input of the Succ Driver, giving a total forward latency of 2 gate delays. In both implementations, the Backward Transfer part is implemented by the Pred Driver of **Figure 4**. There is 1 additional gate between outgoing handshake channel **out[sw]** of the Amplify module and the input of the Pred Driver, giving a total backward latency of 2 gate delays.

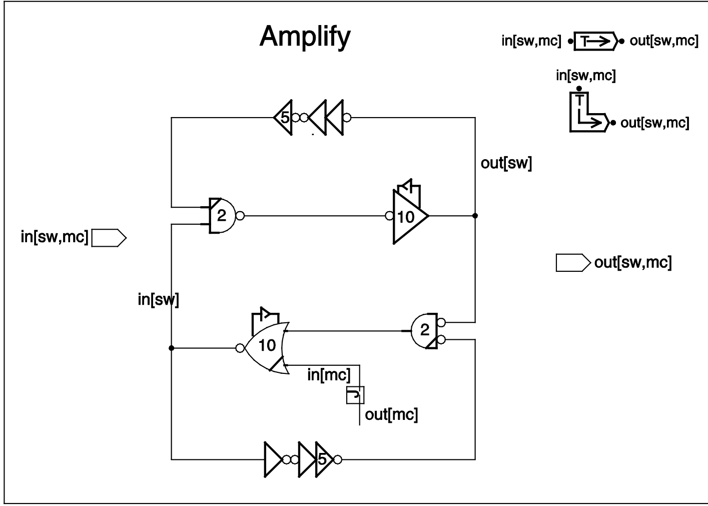


Figure 8: Circuit for Telescope GasP control module Amplify, with input statewire and master clear **in[sw,mc]** and output statewire and joined master clear **out[sw,mc]**. We can use either of the two icons in the top-right corner of this picture to represent an Amplify module. This design is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for in[sw] and out[sw].

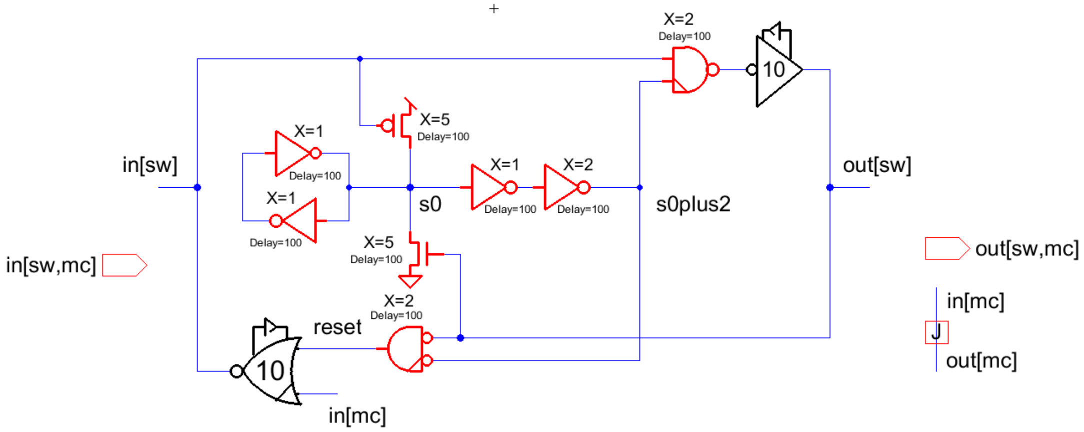


Figure 9: Second Telescope GasP implementation for module Amplify, different than the first one in **Figure 8**. Here, we share the self-resetting loops between incoming and outgoing statewires insofar possible. This second Amplify implementation is saved in the library name TelescopeGasP_wStateSharing, under module name AmplifyT. The appendix `T' in the module name stands for `Telescope'. The design is based on [4].

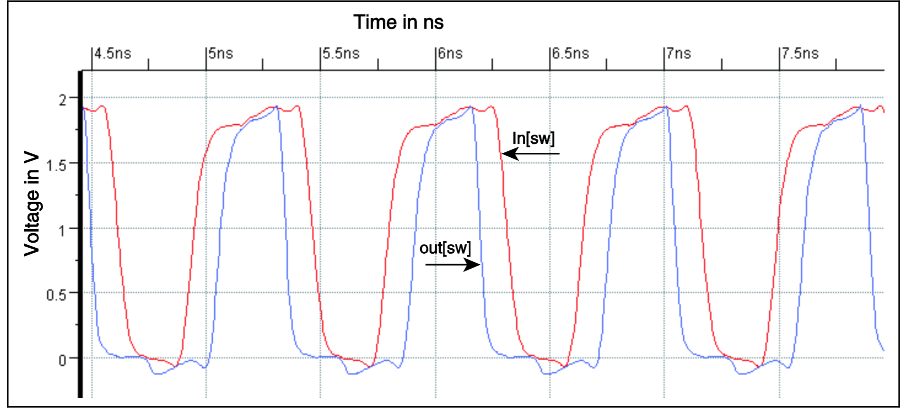


Figure 10: SPICE-level simulation of the Telescope GasP implementation of the Amplify module in **Figure 8**. Red signal in[sw] and blue signal out[sw] have a telescopic handshake relationship: in[sw] HI causes out[sw] to go HI, and when out[sw] has gone LO, only then will in[sw] go LO. The simulation clearly shows that each HI pulse of out[sw] is contained strictly within a HI pulse of in[sw].

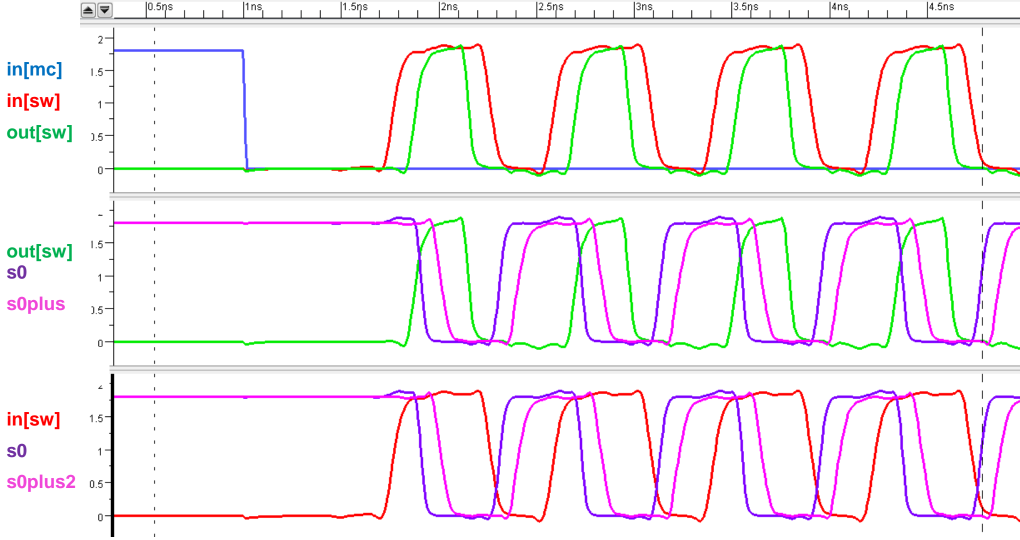


Figure 11: SPICE-level simulation of module Amplify in **Figure 9**.

- The top window clearly shows the telescoping handshake relation between in[sw] and out[sw].
- The middle window shows the simulated delay margins from out[sw] HI in green to s0 LO in purple (1 gate delay) and to s0plus2 LO in pink (3 gate delays) and back to the green out[sw] no-drive and LO (minimal 5 gate delays). The in[sw] reset safety margin of 3 gate delays that we see in **Figure 9** when going from out[sw] HI disabling the inverted input AND gate to s0plus2 LO enabling the AND gate is large enough to guarantee the telescope relation between in[sw] and out[sw]. It is also small enough to fit within the minimum HI pulse width of 5 gate delays for out[sw] in order to maintain throughput. The remaining 2 gate delays after s0plus2 goes LO are used to stop the HI drive on out[sw] (see Figure) so the receiver can reset out[sw] without a fight conflict.
- Similar to the middle window, the bottom window shows the simulated delay margins from in[sw] going LO in red to s0 HI in purple (1 gate delay) and to s0plus2 HI in pink (3 gate delays) and back to the red in[sw] going HI (minimal 5 gate delays). The out[sw] set safety margin of 3 gate delays that we see in **Figure 9** when going from in[sw] LO disabling the NAND gate to s0plus2 HI enabling the NAND gate is large enough to guarantee a 1-1 telescopic handshake relation between in[sw] and out[sw]. It is also small enough to fit within the minimum LO pulse width of 5 gate delays for in[sw] in order to maintain throughput. As can be seen from **Figure 9**, the remaining 2 gate delays after s0plus2 goes HI are used to stop the LO drive on in[sw], so the sender can set in[sw] without a fight conflict.

Without a fast reset signal, the minimum cycle time for each implementation is 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay loops in the circuit diagrams of **Figure 8** and **Figure 9,** assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each additional Amplify module. This skewing of the cycle time happens because each successive Amplify module's Backward Transfer section must wait until the corresponding Forward Transfer in the module has completed.

Using a fast reset signal, we can reduce the backward latency and reduce the cycle time. We designed a test environment that generates such a fast reset signal and that passes it to the Amplify module via the existing master clear signal in the module. We tested the behavior of the Amplify module with this fast reset, using the configuration in **Figure 12**.

Figure 12 shows a sender queue on the left, designed using traditional 6-4 GasP, that feeds a data path consisting of a first-in-first-out queue of three Amplify modules, designed in Telescope GasP using the implementation in **Figure 9**. The right-hand side shows the receiving half of a Store module, designed in 6-4 GasP using the implementation in **Figure 6**. We deleted the sending half from the Store implementation, because we don't need it, and we added extra circuitry to generate a fast reset signal, which is called *mc_fastRTZ*. This new configuration works as follows.

Four gate delays after the start of an incoming handshake to the Store module, i.e. after out[sw] goes HI, the Store module makes mc_fastRTZ become HI. At the same time, i.e. four gate delays after out[sw] goes HI, the Store module makes its local clock signal *cl* HI. The mc_fastRTZ signal is distributed to all Amplify as master clear input to the LO driver for the module's incoming handshake channel. The *cl* signal, besides clocking latches in the data path, also acts as input to the LO driver for out[sw], the handshake channel going out of the last Amplify module in the queue. Consequently, five gate delays after out[sw] goes HI, all outstanding handshake communications in the Amplify queue are reset concurrently. The reset stops after 5 gate delays, via the self-resetting loop in the Store module. **Figure 13** and **Figure 14** show SPICE-level simulation waveforms of mc_fastRTZ, and the corresponding concurrent reset to LO for Amplify channels in[sw], m1[sw], m2[sw], and out[sw].

The cycle time for the Amplify queue without a fast reset signal would be 10 + 3*4 = 22 gate delays. With the fast reset signal, we obtain a cycle time of 10 + 3*2 = 16. These results would be the same had we used the Amplify implementation in **Figure 8**.

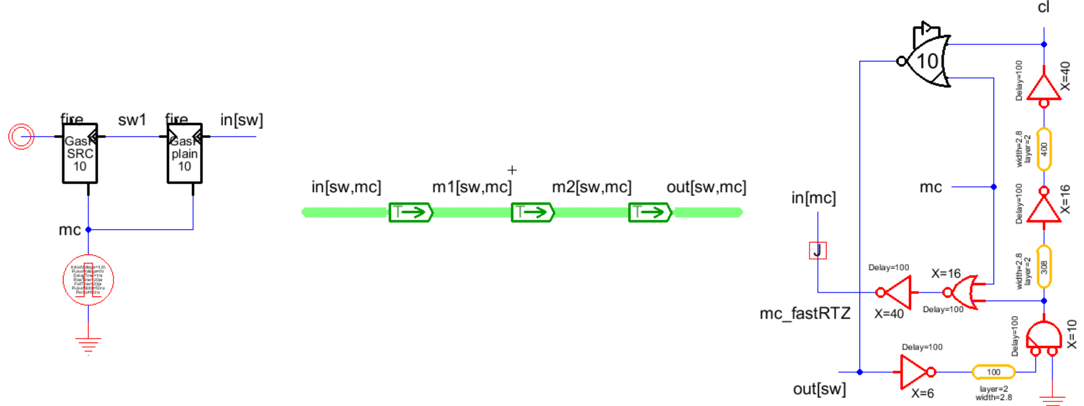


Figure 12: Fast reset configuration and test environment to concurrently reset all handshakes in the three Amplify modules in the middle queue.

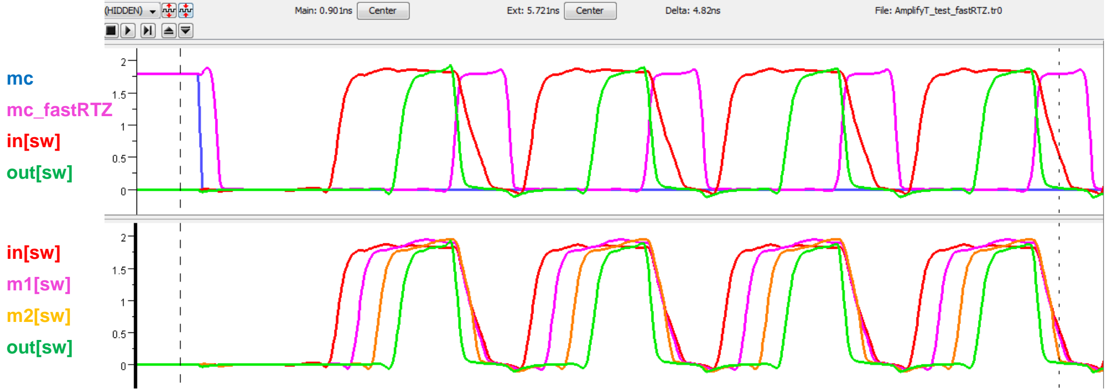


Figure 13: SPICE-level simulation results for the fast reset and test configuration in **Figure 12**.

- The top window shows how the pink HI pulse on the fast reset signal, mc_fastRTZ, concurrently resets both the green handshake on out[sw] of the third Amplify module and the red handshake on in[sw]. The reset slope for out[sw] in green is steeper than the reset slope for in[sw] in red. Details on the reset slope situation follow in **Figure 14** below.
- The bottom window shows the beginning and end of each handshake over the initial, intermediate and final statewires in[sw], m1[sw], m2[sw], and out[sw] in the Amplify queue. The handshakes at the outgoing channel of each module start 2 gate delays after the handshakes at the incoming channel. But thanks to the fast reset signal, mc_fastRTZ, shown in the top window, all outstanding handshakes end at the same time. The steepness of the reset slopes for in[sw], m1[sw], m2[sw], out[sw] depends on how many modules there are between the statewire and the Store module at the end of the data path: the fewer modules in-between, the steeper the slope. A more detailed explanation follows in **Figure 14** below. Differences in slope peter out quickly: the slope differences for the falling transitions on in[sw], m1[sw], and m2[sw] are significantly less prominent.

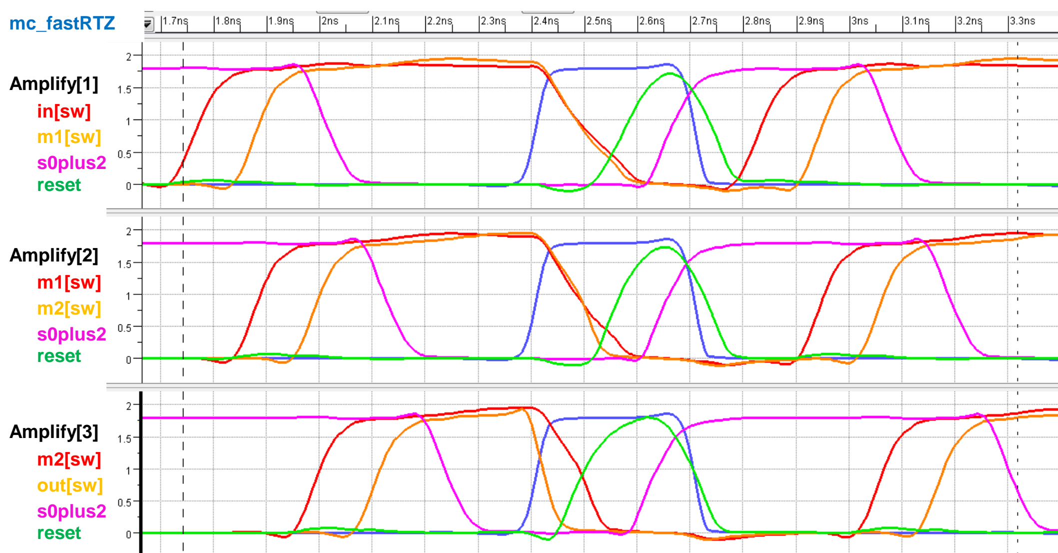


Figure 14: SPICE-level simulation details for the fast reset and test configuration in **Figure 12**. The top window contains waveforms for signals in the first, i.e. left-most Amplify module, Amplify[1], of the 3-stage Amplify queue in **Figure 12**. The middle and bottom windows show the waveforms for similar signals in the second and third Amplify modules in the queue, Amplify[2] and Amplify[3]. The waveforms display the slope differences between the falling handshake transitions on the module's output channel, in orange, and the module's input channel, in red. In each case, the falling slope for the orange signal at the output channel is steeper than the falling slope for the red signal at the input channel. The other signals, in blue, green, and pink, are there to explain why this is so:

- The blue signal, mc_fastRTZ, is the fast reset signal generated by the Store module. During its 5 gate delay HI pulse, all orange and red handshake signals are reset to LO, simultaneously.
- The green signal, reset, is part of the normal backward reset path. When HI, it resets the red incoming handshake signal of the Amplify module to LO, with or without the extra help of mc_fastRTZ. Signal reset is internal to the Amplify module. It rises 1 gate delay after a falling transition on the outgoing handshake signal, as can be seen from the schematics in **Figure 9**.
- From **Figure 12** we can see that the falling handshake for out[sw] is driven from the Pred Driver in the Store module, via the local clock signal *cl*.
  - Signal *cl* is amplified for the purpose of steering data latches, but the test configuration does not include these data latches. As a result, the load on signal cl is much smaller than the load on the fast reset signal, though both are amplified in the same way. This causes *cl* to rise faster than mc_fastRTZ, which in turn causes out[sw] to fall earlier than m2[sw], m1[sw], and in[sw].
  - Also, looking at the implementation of the Pred Driver in **Figure 4**, we can see that *cl* steers an NMOS transistor of strength 10. All other falling handshakes are driven from the Pred Driver in the successor Amplify module in the queue, via the fast reset signal mc_fastRTZ and the master clear driver input. Looking at the implementation of the Pred Driver in **Figure 4**, we can see that the master clear signal steers a much weaker NMOS transistor of strength 3.667. Consequently, out[sw] falls more steeply than m2[sw], m1[sw], and in[sw] - at least initially. As a result, the green reset signal rises first in module Amplify[3], and then in Amplify[2] and then in Amplify[1].
- When the green reset signal rises, it starts helping the blue mc_fastRTZ signal to reset the incoming red handshake signal to LO. This explains the knee and subsequent steeper slope for the red signal in each window. We see such a knee at the initial slope to steeper slope change for m2[w] at about 1 Volt, for m1[sw] at about 0.5 Volt, and for in[sw] at a little below 0.5 Volt. The result is that m2[sw] completes its LO transition before m1[sw], and that both m2[sw] and m1[sw] finish their LO transition before in[sw].
- The pink signal, s0plus2, is part of the self-resetting loop in the Amplify module. It is LO at the start of the mc_fastRTZ HI pulse. It goes HI 4 gate delays after mc_fastRTZ goes HI, and 1 more gate delays later, it stops the HI pulse on the green reset signal. The 5 gate delay HI pulse on the blue mc_fastRTZ signal stops at about the same time. With both Pred Driver inputs LO, the LO drive on the incoming red handshake signal ceases, after a - partially concurrent - drive of 5 gate delays.

5. Fork Modules

Fork modules provide one input to multiple output, or 1-to-NofN, broadcast communication. In Sections 5.1 and 5.2 below, we give Telescope GasP implementations for two-way and three-way Fork modules.

5.1. 2-Way Fork Module, or Fork2

Figure 15 and **Figure 16** show our two Telescope GasP circuit implementation for the 2-way Fork module, or Fork2. The module provides a one input to two output broadcast communication. The Fork2 implementation in **Figure 15** is an example of our first design approach with independent self-resetting loops for the statewires of incoming and outgoing handshake channels. It has two self-resetting loops: the loop at the top serves to start and stop the HI drives for *outA[sw]* and *outB[sw],* the loop at the bottom serves to start and stop the LO drive for *in[sw].* The Fork2 implementation in **Figure 16** is an example of the second design approach which shares self-resetting loops between the statewires of incoming and outgoing handshake channels. It has one self-resetting loop, with separate taps to start and stop the HI drives for *outA[sw]* and *outB[sw]* and the LO drive for *in[sw].*

A SPICE-level simulation of the handshake behavior of the Fork2 implementation in **Figure 15** follows in **Figure 17**. The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **in[sw]**, **outA[sw]**, **outB[sw]**, and **mc** are LO. When **in[sw]** goes HI, indicating the presence of data, and both **outA[sw]** and **outB[sw]** are LO, indicating the availability of broadcast space, the following four actions are started **in sequence**, in the order indicated below:

1. The Forward Transfer part signals new data to both successor modules by driving **outA[sw]** and **outB[sw]** HI. This takes 2 gate delays.
2. The Forward Reset part kicks in, cutting off both HI forward drives after 5 gate delays.
3. The Backward Transfer part waits until both **outA[sw]** and **outB[sw]** are LO, and then reports new availability of space to the predecessor module by driving **in[sw]** LO, 2 gate delays later.
4. The Backward Reset part kicks in, cutting off the LO backward drive after 5 gate delays.

At the end of these four **serial** actions, the Fork2 module waits until **in[sw]** is HI again, so it can start its next cycle.

The waveforms in **Figure 17** support the above behavioral description of the module. We deliberately skewed the loads and delays for **outA[sw]** and **outB[sw]** to make both signals visible. Had we not done so, their pulses would overlap.

Internally, the behaviors of the two Fork2 implementations differ. The advantage of the implementation in **Figure 16** is that it has relatively safe delay margins, whereas the implementation in **Figure 15** has two delay margins that are marginal. The differences in delay margin are very similar to the ones we described in Section 4 for the two implementations of module Amplify.

The two marginal delay margins in **Figure 15** are as follows:

1. After *in[sw]* rises, it takes 2 gate delays before both *outA[sw]* and *outB[sw]* rise and disable the inverted input AND gate in the lower self-resetting loop, while it takes 3 gate delays for the HI *in[sw]* signal to enable that same inverted input AND gate via the lower self-resetting loop. To guarantee a telescope relation between *in[sw]* and *outA[sw]-outB[sw],* the inverted input AND gate must sense the two disabling *outA[sw]* and *outB[sw]* transitions before it senses the enabling *in[sw]* transition. The margin between the first two transitions and the latter is only 1 gate delay.
2. After both *outA[sw]* and *outB[sw]* have fallen, it takes 2 gate delays before *in[sw]* falls and disables the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the LO *outA[sw]* and *outB[sw]* signals to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *in[sw]* and *outA[sw]-outB[sw],* the NAND gate must sense the disabling *in[sw]* before it senses the enabling *outA[sw]* and *outB[sw]* transitions. The delay margin between the disabling and the two enabling transitions is only1 gate delay.

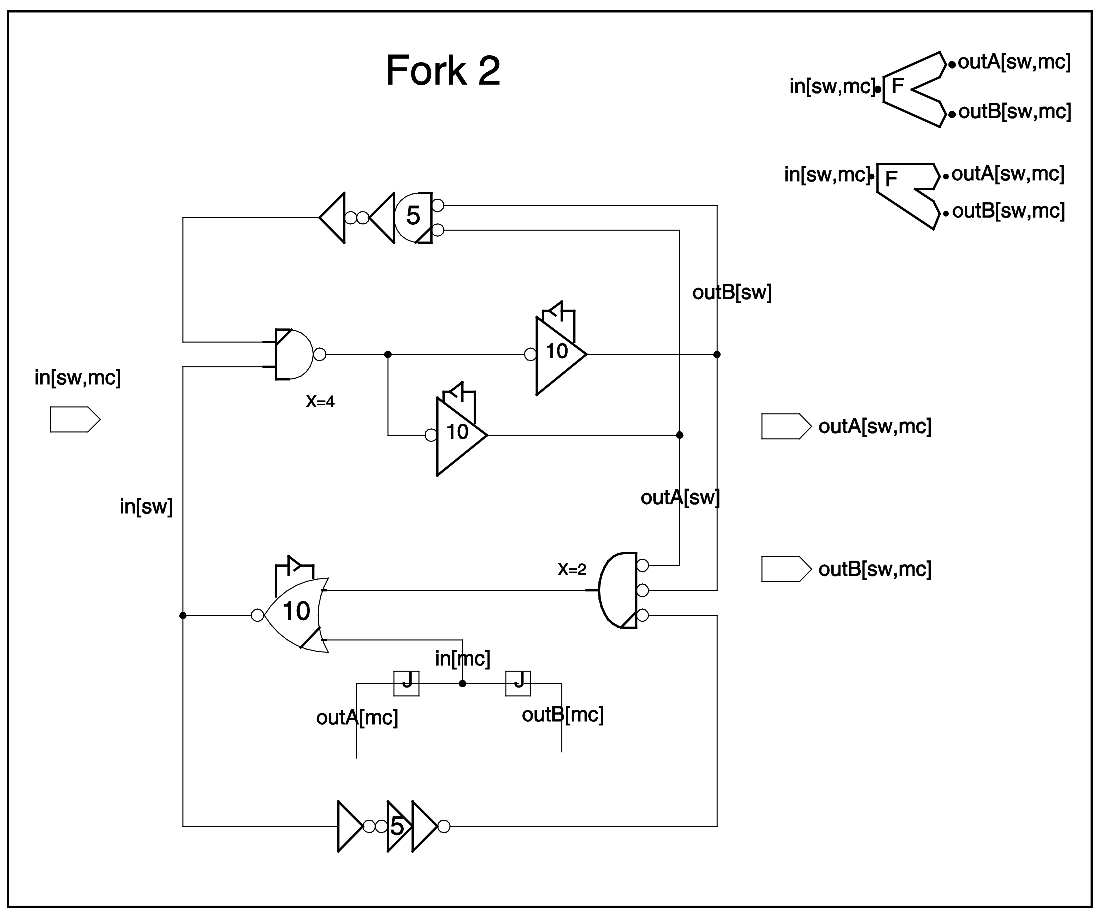


Figure 15: Circuit for Telescope GasP control module Fork2, which is a 2-way Fork. It has input statewire and master clear **in[sw,mc]** and two broadcast output statewires with a joined master clear, **outA[sw,mc]** and **outB[sw,mc]**. We will use either of the two icons in the top-right corner of this picture, when we represent this a 2-way Fork module. This design is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for in[sw] and outA[sw]-outB[sw].

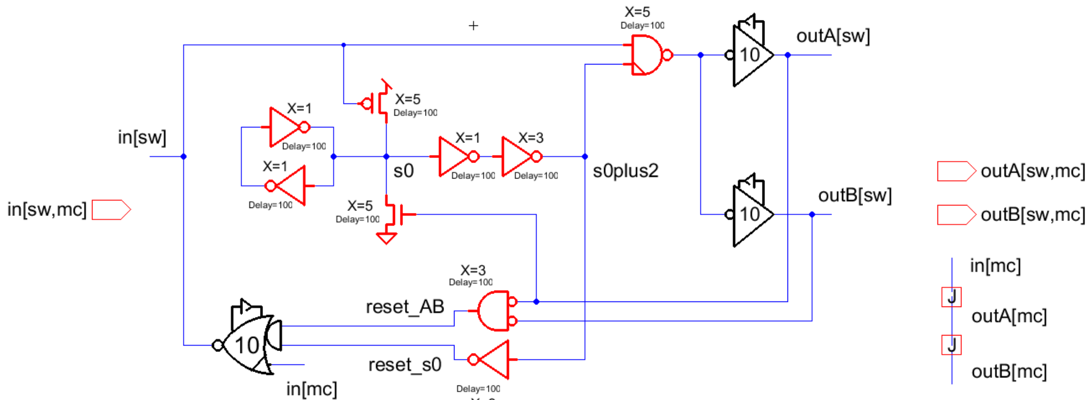


Figure 16: Second Telescope GasP implementation for module Fork2, different from the first one in **Figure 15**. This design shares the self-resetting loops between incoming and outgoing statewires insofar as possible. It is saved in library TelescopeGasP_wStateSharing, under module name Fork2T (appendix `T' in the module name stands for `Telescope').

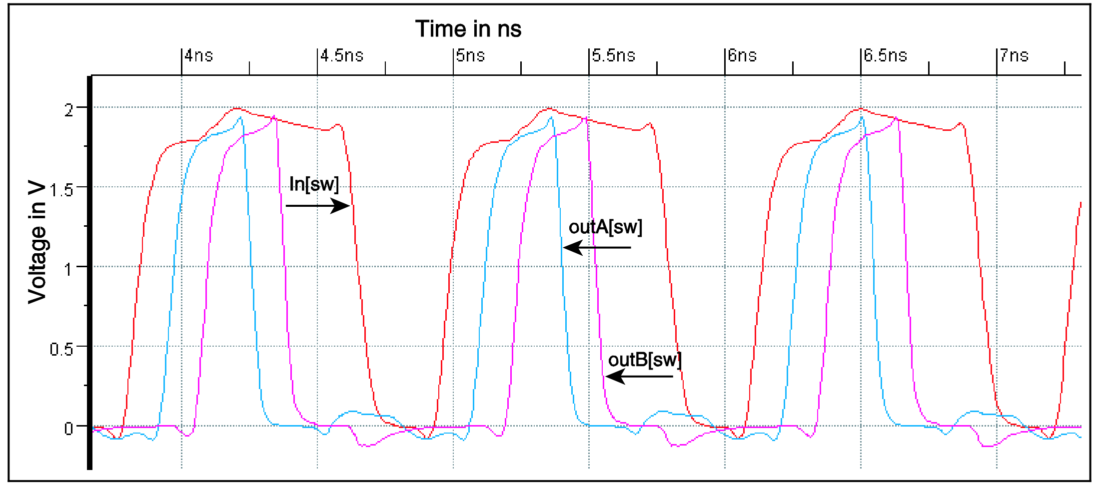


Figure 17: SPICE-level simulation of the Telescope GasP implementation of the 2-way Fork module. We deliberately skewed the loads and delays for the blue and pink signals outA[sw] and outB[sw] to separate their pulses. Red signal in[sw] has a telescopic handshake relation with blue and pink signals outA[sw] and outB[sw]: in[sw] HI causes both outA[sw] and outB[sw] to go HI, and when both outA[sw] and outB[sw] have gone LO again only then will in[sw] go LO also. The simulation clearly shows that both HI pulses for outA[sw] and outB[sw] are strictly contained within the HI pulse for in[sw].

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Fork2 module in **Figure 15** as safe as those in **Figure 16.**

The two implementations have the same cycle time and latencies. In both implementations, the two Forward Transfer parts are implemented using the Succ Driver of **Figure 2**. There is 1 additional gate between **in[sw]** and the input of the two Succ Drivers, giving a total forward latency of 2 gate delays. In both implementations, the Backward Transfer part is implemented by the Pred AND Driver of **Figure 5**. There is 1 additional gate between the two outgoing channels **outA[sw]** and **outB[sw]** of the Fork2 module and the input of the Pred AND Driver, giving a total backward latency of 2 gate delays.

Without a fast reset signal, the minimum cycle time for each implementation is 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops in the circuit diagrams of **Figure 15** and **Figure 16,** assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each successor module between the Fork2 module and the following Store module. This skewing of the cycle time happens because each next module's Backward Transfer section must wait until the corresponding Forward Transfer in the module has completed.

Using a fast reset signal, we can reduce the backward latency and lower the cycle time. We designed a test environment that generates such a fast reset signal and that passes it to the Fork2 module via the existing master clear signal in the module. We tested the behavior of the Fork2 module with this fast reset, using the configuration in **Figure 18**.

Figure 18 shows a sender queue on the left, designed using traditional 6-4 GasP, that feeds a data path consisting of a tree of Fork2 modules with maximum depth of 3, designed in Telescope GasP using the implementation in **Figure 16**. The right-hand side shows the receiving half of a Store module, designed in 6-4 GasP using the implementation in **Figure 6**. We deleted the sending half from the Store implementation, because we don't need it, and we added extra circuitry to generate a fast reset signal, which is called *mc_fastRTZ,* and to reset all outgoing handshakes at the end of the Fork2 tree. This new configuration works in a way similar to the fast reset configuration for the Amplify module in **Figure 12**.

Four gate delays after the last incoming handshake to the Store module, i.e. after outA[sw], outB[sw], outC[sw], outD[sw], and outE[sw] have gone HI, the Store module makes mc_fastRTZ HI. At the same time, i.e. four gate delays after outA[sw] to outE[sw] have gone HI, the Store module makes its local clock signal *cl* HI. The mc_fastRTZ signal is distributed to each Fork2 module as master clear input to the LO driver for the module's incoming handshake channel. The *cl* signal, besides clocking latches in the data path, also acts as input to the LO driver for outA[sw] to outE[sw], the handshake channels going out of the last Fork2 modules in the tree. Consequently, five gate delays after outA[sw] to outE[sw] have gone HI, all outstanding handshake communications in the Fork2 tree are reset concurrently. The reset stops after 5 gate delays, via the self-resetting loop in the Store module.

Figure 19 and **Figure 20** show SPICE-level simulation waveforms of mc_fastRTZ, and the corresponding concurrent reset to LO for Fork2 channels in[sw], m1[sw], m2[sw], and outA[sw].

The cycle time for the Fork2 tree without a fast reset signal would be 10 + 3*4 = 22 gate delays. With the fast reset signal, we obtain a cycle time of 10 + 3*2 = 16. These results would be the same had we used the Fork2 implementation in **Figure 15**.

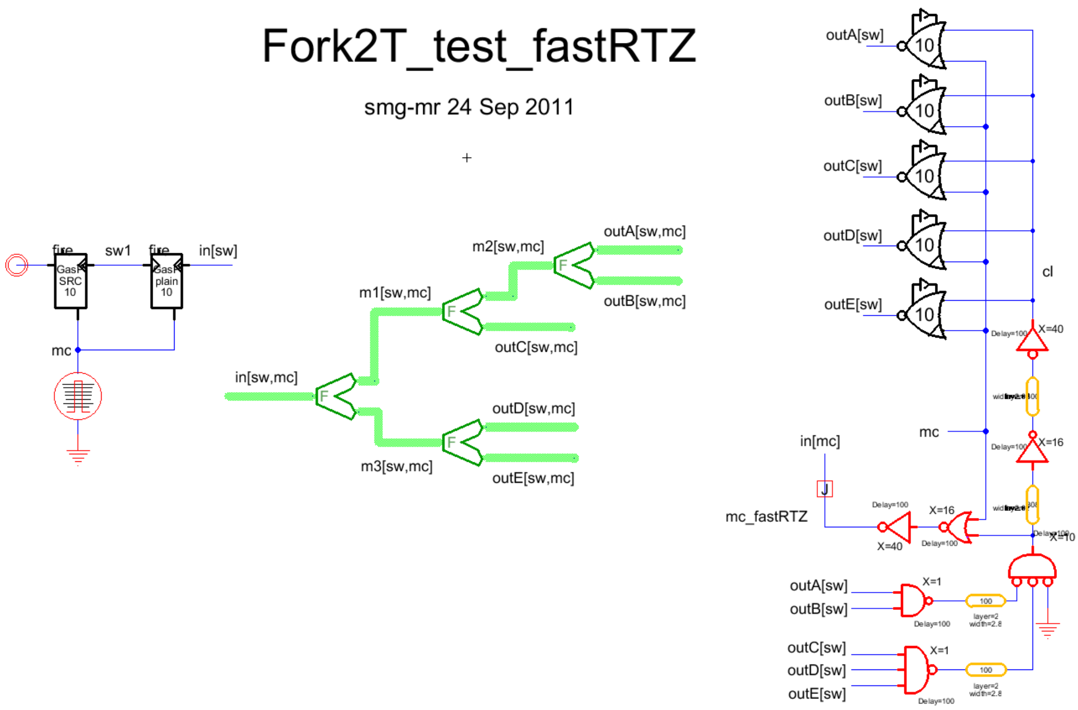


Figure 18: Fast reset configuration and test environment to reset concurrently all handshakes in the Fork2 modules in the middle tree. Note that this is a much more complex configuration than the one in **Figure 12** that concurrently resets the queued Amplify modules. It is also less modular than we would like it to be. We anticipate that, in practice, outA[sw] to outE[sw] are reset either by individual Store modules, or by some re-converging tree that joins these final statewires, thus maintaining modularity. We also anticipate that, in practice, we will be able to identify a particular Store module from which we can tap the mc_fastRTZ signal.

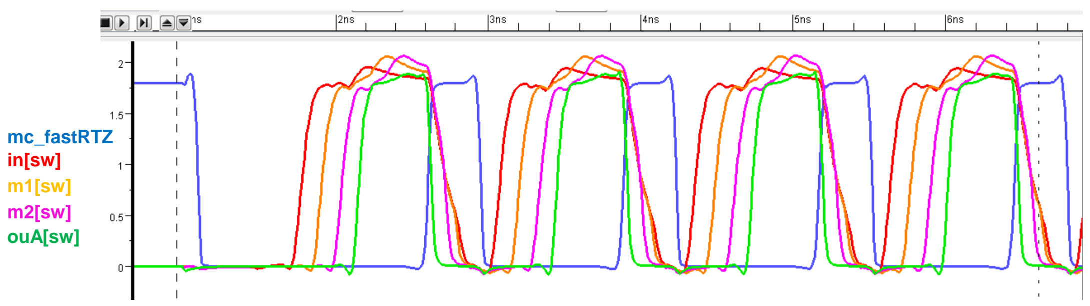


Figure 19: SPICE-level simulation results for the fast reset and test configuration in **Figure 18**. Similar to the fast reset results for module Amplify, here we see the beginning and end of each handshake over the initial, intermediary and final statewires in[sw], m1[sw], m2[sw], and outA[sw] in the Fork2 tree. The handshakes at the outgoing channel of each module start 2 gate delays after the handshakes at the incoming channel. The blue HI pulse on the fast reset signal, mc_fastRTZ, concurrently resets all statewires from outA[sw] to in[sw]. The reset slope is steepest for the green-colored statewire outA[sw], then for pink statewire m2[sw], followed by orange statewire m1[sw], and it is shallowest for the red-colored statewire in[sw]. A more detailed explanation follows in **Figure 20** below.

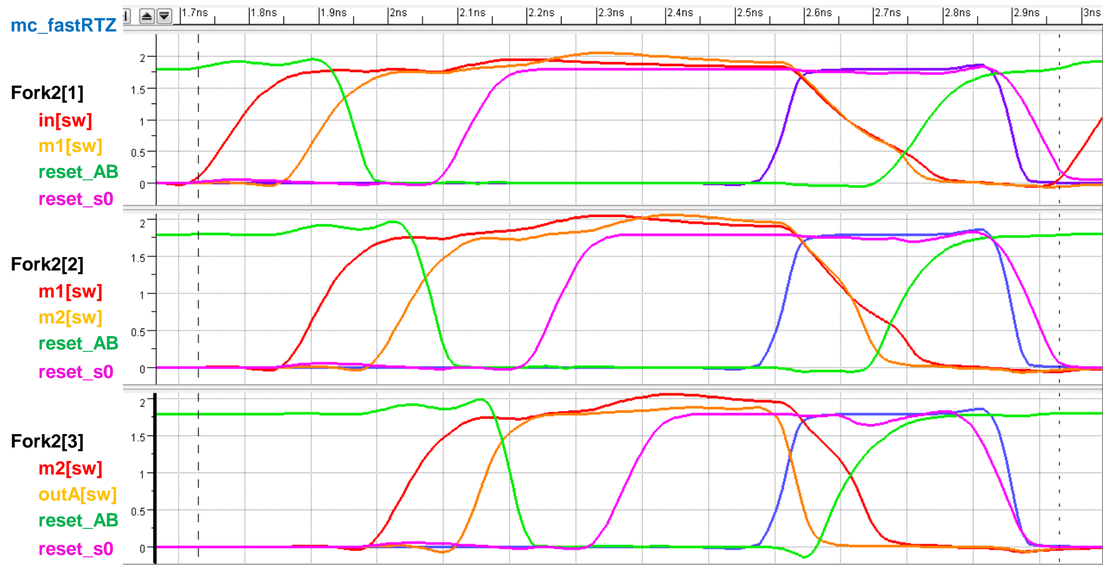


Figure 20: SPICE-level simulation details for the fast reset and test configuration in **Figure 18**. The top window contains waveforms for signals in the left-most Fork2 module Fork2[1]. The middle and bottom windows show the waveforms for similar signals in the following top-most second and third Fork2 modules, Fork2[2] and Fork2[3]. The details are very similar to the fast reset details for module Amplify in **Figure 14**.

- The green signal, reset_AB, is part of the normal backward reset path. When HI, it resets the red incoming handshake signal of the Fork2 module to LO, provided reset_s0 is also HI. The two AND-ed resets work with or without extra help of mc_fastRTZ. Signal reset_AB is internal to the Fork2 module. It rises 1 gate delay after both outgoing handshakes have fallen – see the schematics in **Figure 16**.
- The falling handshake for outA[sw] is driven from the Pred Driver in the Store module, via the local clock signal *cl*.
- From **Figure 12** we can see that the falling handshake for outA[sw] is driven from the Pred Driver in the Store module, via the local clock signal *cl*.
  - Signal *cl* is amplified for the purpose of steering data latches, but the test configuration does not include these data latches. As a result, the load on signal cl is much smaller than the load on the fast reset signal, though both are amplified in the same way. This causes *cl* to rise faster than mc_fastRTZ, which in turn causes outA[sw] to fall earlier than m2[sw], m1[sw] and in[sw].
  - Also, looking at the implementation of the Pred Driver in **Figure 4**, we can see that *cl* steers an NMOS transistor of strength 10. In contrast, the falling handshakes for m2[sw], m1[sw], and in[sw] are driven from the Pred AND Driver in the successor Fork2 modules in the queue, via the fast reset signal mc_fastRTZ and the master clear driver input. From the Pred AND Driver in **Figure 5**, we can see that the master clear signal steers a much weaker NMOS transistor of strength 3.667. Consequently, outA[sw] falls more steeply than m2[sw], m1[sw], and in[sw] - at least initially. As a result, the green reset signal rises first in module Fork2[3], and then in Fork2[2], and then in Fork2[1].
- When the green reset signal rises, it starts helping the blue mc_fastRTZ signal in resetting the incoming red handshake signal to LO. This explains the knee and subsequent steeper slope for the red signal in each window. We see such a knee at the initial slope to steeper slope change for m2[w] at about 1 Volt, for m1[sw] at a little above 0.5 Volt, and for in[sw] at a little below 0.5 Volt. The result is that the LO transition completes first for m2[sw] then for m1[sw], and last for in[sw].
- The pink signal, reset_s0, is a 1 gate delay inverted s0plus2 signal, and is part of the self-resetting loop in the Fork2 module. It is HI at the start of the mc_fastRTZ HI pulse. It goes LO 5 gate delays after mc_fastRTZ goes HI, stopping the LO drive through the AND gate that it drives jointly with reset_AB. The 5 gate delay HI pulse on mc_fastRTZ stops at about the same time. With all Pred AND Driver inputs LO, the LO drive on the incoming red statewire ceases, after a - partially concurrent - drive of 5 gate delays.

5.2. 3-Way Fork Module, or Fork3

Figure 21 and **Figure 22** show our two Telescope GasP circuit implementations for the 3-way Fork module, called **Fork3**. The designs are very similar to the two implementations of the 2-way Fork module in **Figure 15** and **Figure 16**. From these 2-way to 3-way Fork extensions one can already see the limit in implementing an N-way Fork for the same latency and throughput as a 2-way Fork, for large N. All N signals must be synchronized at the NAND and negated-input AND gates at the start of the Forward and Backward Transfer parts. Beyond N=3, we might want to build trees of Fork modules.

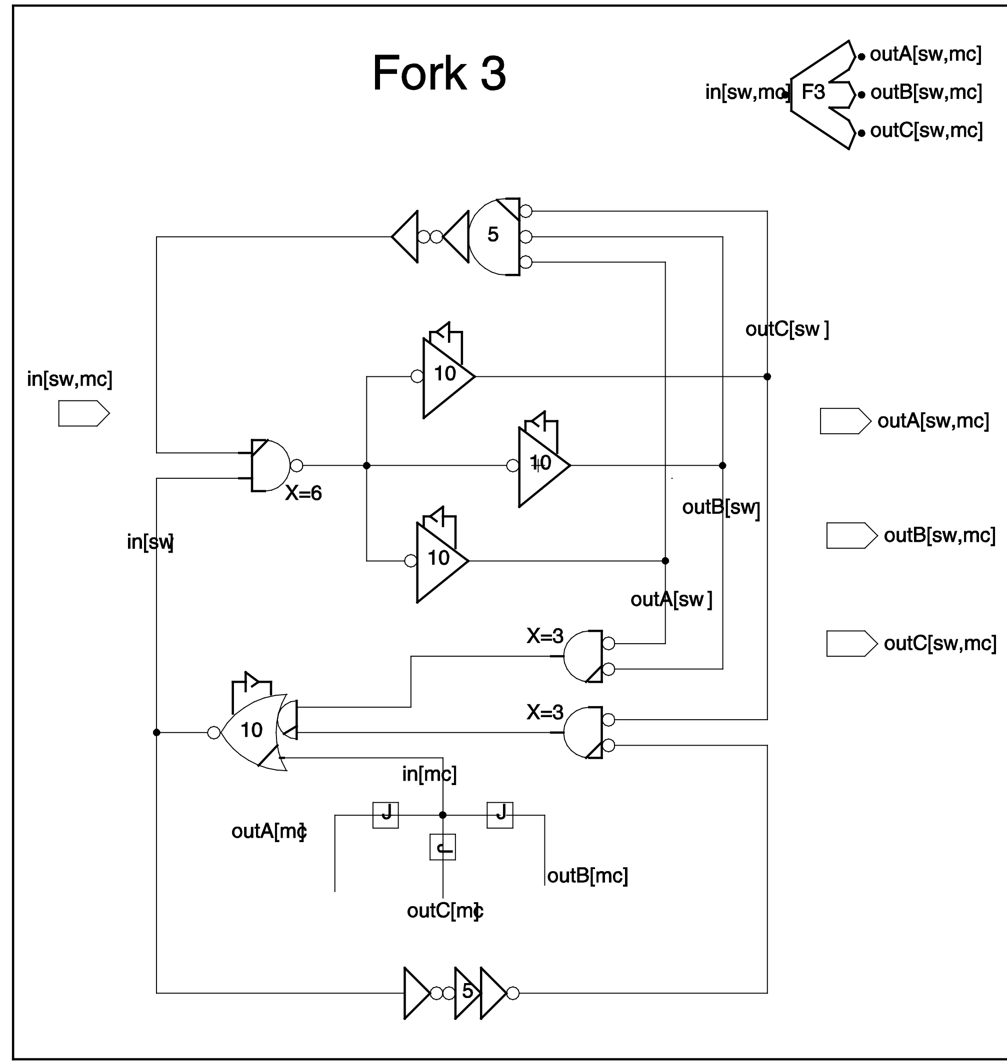


Figure 21: Circuit for Telescope GasP control module Fork3, which is a 3-way Fork. It has input statewire and master clear **in[sw,mc]** and three broadcast output statewires with a joined master clear, **outA[sw,mc]**, **outB[sw,mc]**, and **outC[sw,mc]**. We will use the top-right icon when we represent a 3-way Fork module. This design is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for in[sw] and outA[sw]-outB[sw]-outC[sw].

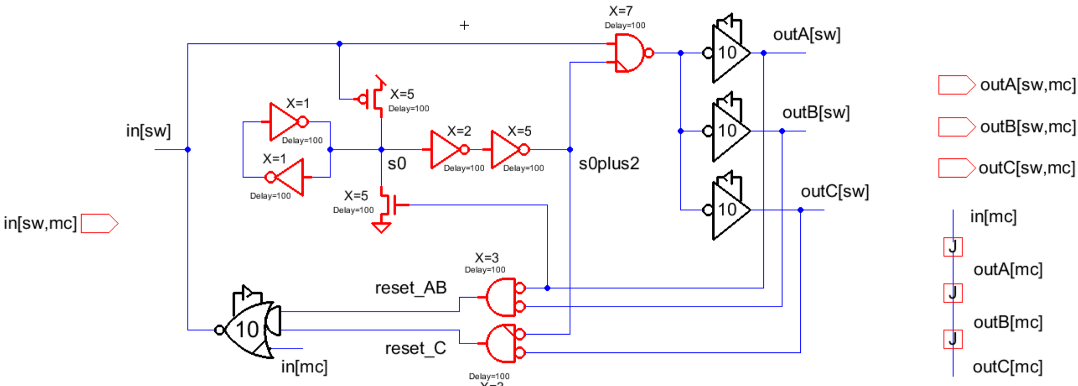


Figure 22: Second Telescope GasP implementation for module Fork3, different than the first one in **Figure 21**. Here, we share the self-resetting loops between incoming and outgoing statewires insofar possible. This second Fork3 implementation is saved in the library name TelescopeGasP_wStateSharing, under module name Fork3T (appendix `T' in the module name stands for `Telescope').

6. Join Modules

Join modules provide many-input-to-one-output broadcast synchronization. In Sections 6.1 and 6.2 below, we will give Telescope GasP implementations for two-way and three-way Join modules.

6.1. 2-Way Join Module, or Join2

Figure 23 and **Figure 24** show our Telescope GasP circuit implementation for the 2-way Join module, or **Join2**. Join2 is more or less the reverse of Fork2. The module merges two incoming communication streams into one output communication stream. The Join2 implementation in **Figure 23** is an example of our first design approach with independent self-resetting loops for statewire of incoming and outgoing handshake channels. It has two self-resetting loops: the loop at the top is used to start and stop the HI drive for *out[sw],* the loop at the bottom is used to start and stop the LO drives for *inA[sw]* and *inB[sw].* The Join2 implementation in **Figure 24** is an example of the second design approach where we share self-resetting loops between the statewires of incoming and outgoing handshake channels, where possible. It has one self-resetting loop, with separate taps to start and stop the HI drive for *out[sw]* and the LO drives for *inA[sw]* and *inB[sw].*

A SPICE-level simulation of the handshake behavior of the Fork2 implementation in **Figure 23** follows in **Figure 25**. The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **inA[sw]**, **inB[sw]**, **out[sw]**, and **mc** are LO. When **inA[sw]** and **inB[sw]** go HI, indicating the broad presence of data on both incoming channels, and when **out[sw]** is LO, indicating the availability of space, the following four sets of actions are started **in sequence**, in the order indicated below:

1. The Forward Transfer part signals new data to the successor module by driving **out[sw]** HI. This takes 2 gate delays.
2. The Forward Reset part kicks in, cutting off the HI forward drives after 5 gate delays.
3. The Backward Transfer parts waits until **out[sw]** is LO, and then report new availability of space to the predecessor modules by driving both **inA[sw]** and **inB[sw]** LO, 2 gate delays later.
4. The Backward Reset part kicks in, cutting off both LO backward drives after 5 gate delays.

At the end of these four **serial** sets of actions, the Join2 module waits until **inA[sw]** and **inB[sw]** are HI again, so it can start its next cycle.

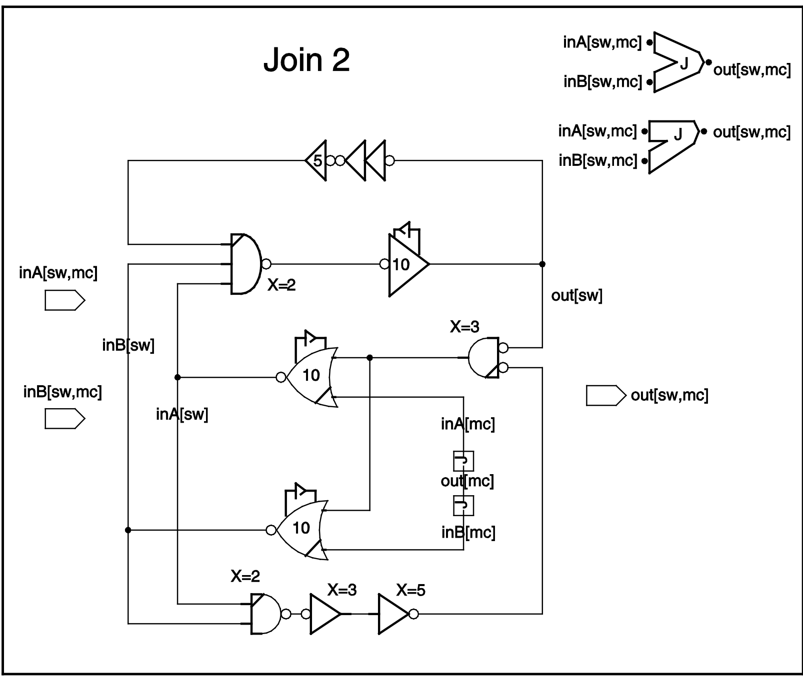


Figure 23: Circuit for Telescope GasP control module Join2, which is a 2-way Join. It has two input statewires with a master clear signal, **inA[sw,mc]** and **inB[sw,mc]**, and one output statewire with a joined master clear, **out[sw,mc]**. We will use either of the two icons in the top-right corner when we represent a 2-way Join module. This design is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for inA[sw]-inB[sw] and out[sw].

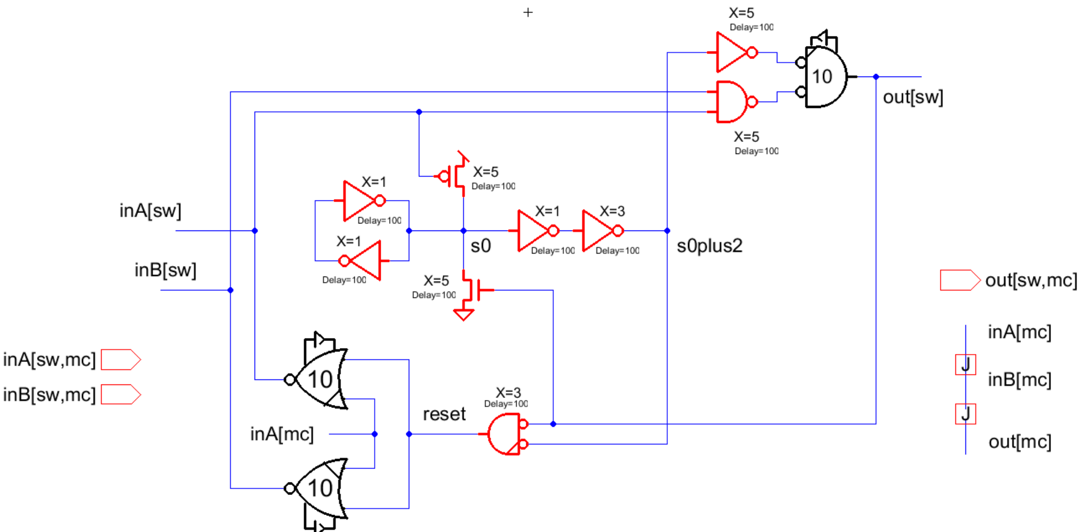


Figure 24: Second Telescope GasP implementation for module Join2. Here, we share the self-resetting loops between incoming and outgoing statewires insofar as possible. This implementation is saved in the library name TelescopeGasP_wStateSharing, under module name Join2T (appendix `T' stands for `Telescope').

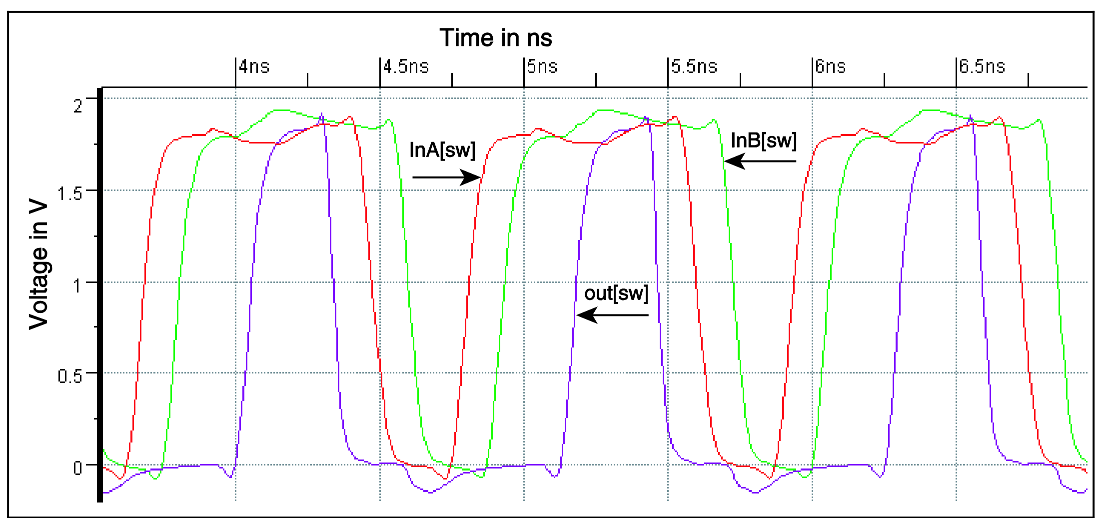


Figure 25: SPICE-level simulation of the Telescope GasP implementation of Join2 in **Figure 23**. We deliberately skewed the loads and delays for the red and green signals inA[sw] and inB[sw] to separate their pulses. Red and green signals inA[sw] and inB[sw] have a telescopic handshake relationship with blue signal out[sw]: inA[sw] HI and inB[sw] HI causes out[sw] to go HI, and when out[sw] has gone LO again only then will inA[sw] and inB[sw] go LO also. The simulation clearly shows that the HI pulse for out[sw] is strictly contained within the HI pulses for inA[sw] and inB[sw].

The waveforms in **Figure 25** support the above behavioral description of the module. We deliberately skewed the loads and delays for **inA[sw]** and **inB[sw]** to make both signals visible. Had we not done so, their pulses would overlap.

Internally, the behaviors of the two Join2 implementations differ. The advantage of the implementation in **Figure 24** is that it has delay margins that are relatively safe, whereas the implementation in **Figure 23** has two delay margins that are marginal. The differences in delay margin are very similar to the ones we described earlier for the two implementations of other broadcast modules, Amplify respectively Fork2.

The two marginal delay margins in **Figure 23** are as follows:

1. After *inA[sw]* and *inB[sw]* have both risen, it takes 2 gate delays before *out[sw]* rises and disables the inverted input AND gate in the lower self-resetting loop, while it takes 3 gate delays for the HI *inA[sw]* and *inB[sw]* signals to enable that same inverted input AND gate via the lower self-resetting loop. To guarantee a telescope relation between *inA[sw]-inB[sw]* and *out[sw],* the inverted input AND gate must sense the disabling *out[sw]* transition before it senses the enabling *inA[sw]* and *inB[sw]* transitions. The margin between disabling and enabling is only 1 gate delay.
2. After *out[sw]* has fallen, it takes 2 gate delays before both *inA[sw]* and *inB[sw]* fall and disable the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the LO *out[sw]* signal to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *inA[sw]-inB[sw]* and *out[sw],* the NAND gate must sense the disabling transitions coming from *inA[sw]* and *inB[sw]* before it senses the enabling *out[sw]* transition. The margin between disabling and enabling is only 1 gate delay.

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Fork2 module in **Figure 23** as safe as those in **Figure 24**.

The two implementations have the same cycle time and latencies. In both implementations, the Forward Transfer part is implemented using the Succ Driver of **Figure 2**. There is 1 additional gate between **inA[sw]** respectively **inB[sw]** and the input of the Succ Driver, giving a total forward latency of 2 gate delays. In both implementations, the two Backward Transfer parts are implemented using the Pred AND Driver of **Figure 5**. There is 1 additional gate between the outgoing channel **out[sw]** of the Join2 module and the input of the two Pred AND Drivers, giving a total backward latency of 2 gate delays.

Without a fast reset signal, the minimum cycle time for each implementation is 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops in the circuit diagrams of **Figure 23** and **Figure 24**, assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each successor module between the Join2 module and the following Store module. This skewing of the cycle time happens because each next module's Backward Transfer section must wait until the corresponding Forward Transfer in the module has completed.

Using a fast reset signal, we can reduce the backward latency and lower the cycle time. We designed a test environment that generates such a fast reset signal and that passes it to the Fork2 module via the existing master clear signal in the module. We tested the behavior of the Join2 module with this fast reset, using the configuration in **Figure 26**.

Figure 26 shows a set of sender queues on the left, designed using traditional 6-4 GasP, that feed a data path consisting of a tree of Join2 modules with a depth of 3, designed in Telescope GasP using the implementation in **Figure 24**. The right-hand side shows the receiving half of a Store module, which matches that in **Figure 12**. We deleted the sending half from the Store implementation, because we don't need it, and we added extra circuitry to generate a fast reset signal, which is called *mc_fastRTZ,* and to reset the outgoing handshake at the end of the Join2 tree. This new configuration works in a way similar to the fast reset configurations for Amplify and Fork2 in **Figure 12** and **Figure 18**, respectively.

Four gate delays after the last incoming handshake to the Store module, i.e. after out[sw] has gone HI, the Store module makes mc_fastRTZ HI. At the same time, i.e. four gate delays after out[sw] has gone HI, the Store module makes its local clock signal *cl* HI. The mc_fastRTZ signal is distributed to each Join2 module as master clear input to the LO driver for the module's incoming handshake channel. The *cl* signal, besides clocking latches in the data path, also acts as input to the LO driver for out[sw]. Consequently, five gate delays after out[sw] has gone HI, all handshake communications in the Join2 tree are reset concurrently. The reset stops after 5 gate delays, via the self-resetting loop in the Store module. **Figure 27** and **Figure 28** show the corresponding SPICE-level simulation waveforms.

Like the earlier 3 deep queue of Amplify modules and the 3 deep tree of Fork modules, the cycle time for the 3 deep tree of Join2 modules without a fast reset signal would be 10 + 3*4 = 22 gate delays. With the fast reset signal, we obtain a cycle time of 10 + 3*2 = 16. These results would be the same had we used the Join2 implementation in **Figure 23**.

Unlike the earlier fast reset simulation results for Amplify and Fork modules, the detailed SPICE simulation in **Figure 28** shows, for the first time that the internal reset signal overlaps with the next handshake cycle. This is the result of (a) a faster reset of the incoming handshake on inD[sw] for the left-most Join2 component in **Figure 26**, Join2[1], and a correspondingly faster start of the next handshake on inD[sw], versus (b) a slower reset of the outgoing handshake on in2[sw] for Join2[1], and a correspondingly slower termination of the internal green reset. We conclude from this that it is probably not a good idea to re-use the weak drive of the master clear signal for fast resets. In the future, we will use a separate fast reset input with a strong first order fast reset drive and an equally strong second-order internal reset drive.

Join2T_test_fastRTZ

smg-mr 24 Sep 2011

sw1 inA[sw] sw4 inC[sw] mc sw2 inB[sw] sw3 inD[sw]

inA[sw,mc] out[sw,mc] inB[sw,mc] m1[sw,mc] inC[sw,mc] inD[sw,mc] m2[sw,mc]

cl inA[mc] mc mc_fastRTZ out[sw]

Figure 26: Fast reset configuration and test environment to concurrently reset all handshakes in the three Join2 modules in the middle tree.

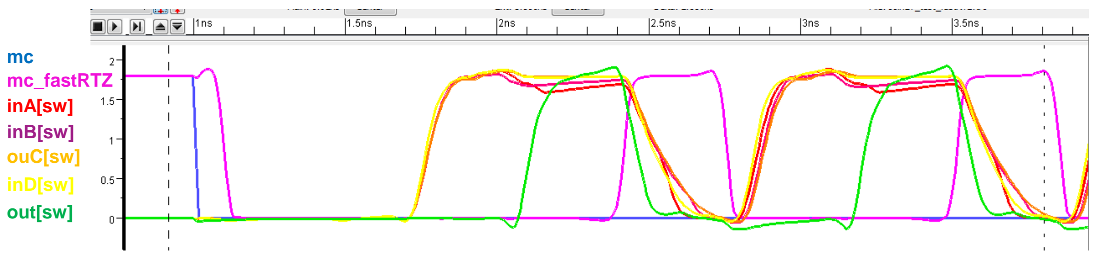


Figure 27: SPICE-level simulation results for the fast reset and test configuration in **Figure 26**. We see the beginning and end of each handshake on incoming channels inA[sw] in red, inB[sw] in purple, inC[sw] in orange, and inD[sw] in yellow, and on green outgoing channel out[sw] in the Join2 tree. The handshake at the outgoing channel of each module starts 2 gate delays after the handshakes at the incoming channels. Given that the tree is three Join2 modules deep, and that the incoming handshakes start at the same time, the handshakes on out[sw] start 6 gate delays after the handshakes on inA[sw] to inD[sw]. The pink HI pulse on the fast reset signal, mc_fastRTZ, concurrently resets all these handshakes. The reset slope starts earlier and is steepest for statewire out[sw]. We will look into the reset slopes in more detail in **Figure 28** below. Note that the 5 gate delay pulse on mc_fastRTZ ends just in time to avoid a fight conflict with the next handshake cycle.

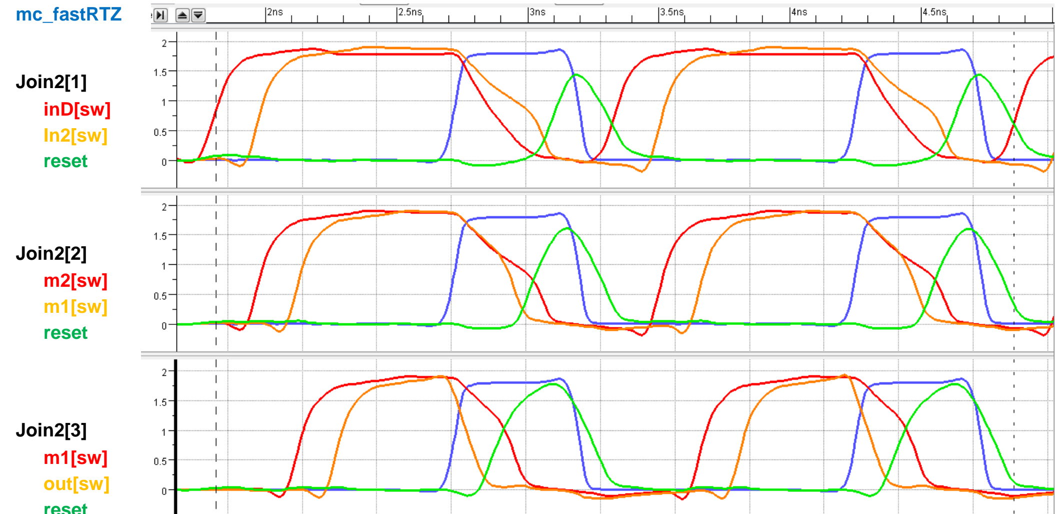


Figure 28: SPICE-level simulation details for the fast reset and test configuration in **Figure 26**. The top window contains waveforms for signals in the first, i.e. left-most Join2 module, Join2[1]. The middle and bottom windows show the waveforms for similar signals in the second and third level Join2 modules. Join2[2] and Join2[3]. The waveforms display the slope differences between the falling handshake transitions on the module's output channel, in orange, and the module's input channel, in red. Unlike the previous fast reset simulations for modules Amplify and Fork2, the falling slope for the orange signal at the output channel is NOT ALWAYS steeper than the falling slope for the red signal at the input channel. Specifically, for Join2[1] the incoming red channel falls more steeply than the outgoing orange channel. Below, we explain why this is so:

- The green signal, reset, is part of the normal backward reset path. When HI, it resets the red incoming handshake signal of the Join2 module to LO. It rises 1 gate delay after the outgoing handshake has fallen, as can be seen from the schematics in **Figure 24**.
- The falling handshake for out[sw] is driven from the Pred Driver in module Store, via local clock signal *cl*.
    - Signal *cl* is amplified for the purpose of steering data latches, but the test configuration does not include these data latches. As a result, the load on signal cl is much smaller than the load on the fast reset signal, though both are amplified in the same way. This causes *cl* to rise faster than mc_fastRTZ, which in turn causes out[sw] to fall earlier than m1[sw], m2[sw], and inD[sw].
    - Also, looking at the implementation of the Pred Driver in **Figure 4**, we can see that *cl* steers an NMOS transistor of strength 10. In contrast, the falling handshakes for m1[sw], m2[sw], and inD[sw] are driven from the Pred Driver in the successor Join2 module in the queue, via the fast reset signal mc_fastRTZ and the master clear driver input. From the Pred Driver in **Figure 4**, we can see that the master clear signal steers a much weaker NMOS transistor of strength 3.667. Consequently, out[sw] falls more steeply than m1[sw], m2[sw], and inD[sw], at least initially. As a result, the green reset signal rises first in module Fork2[3], then in Fork2[2] and last in Fork2[1].
- When the green reset signal rises, it starts helping the blue mc_fastRTZ signal in resetting the incoming red handshake signal to LO. This explains the knee and subsequent steeper slope for the red signal in the bottom two windows. We see such a knee for m1[w] at about 1 Volt and for m2[sw] at 0.8 Volt. For inD[sw] in the top window, the green reset signal signal comes too late: inD[sw] is already LO.
- To understand why the falling slope for the incoming handshake on inD[sw] is steeper than for the outgoing handshake on m2[sw], we must take a closer look at the driver ends of the two statewires. Join2 statewires in1[sw] and in2[sw] are connected to a Succ AND Driver in the predecessor Join2 module. From the Succ AND Driver schematics in **Figure 3**, we can see that this means that these statewires are connected to a PMOS transistor of size 20. In contrast, statewire inD[sw] is connected to a Succ Driver in the 6-4 GasP Store module that acts as its predecessor. From **Figure 2**, we can see that this is a connection to a PMOS transistor of size 10, i.e. half the drain capacitance compared to the PMOS driver connection for in1[sw] and in2[sw]. All other connections to these statewires are similar in size. So, all with all, statewire inD[sw] has a sufficiently lower stray capacitance than statewires in1[sw] and in2[sw] to achieve a steeper falling slope – sufficiently steeper to stand out in the top simulation.
- What also stands out in the top simulation window is that the green internal reset pulse continues beyond the blue fast reset pulse. In the top window, this creates the additional side effect that the green pulse is still driving the LO driver for statewire inD[sw] when inD[sw] is pulled HI to start the next handshake. This is the result of (a) a faster reset of the incoming handshake on inD[sw] and a correspondingly faster start of the next handshake on inD[sw], versus (b) a slower reset of the outgoing handshake on in2[sw] and a correspondingly slower termination of the internal green reset.

6.2. 3-Way Join Module, or Join3

Figure 29 and **Figure 30** show our two Telescope GasP circuit implementations for the 3-way Join module, called **Join3**. The designs are very similar to the two implementations of the 2-way Join module in **Figure 23** and **Figure 24**. Similar to the Fork extensions, one can already see the limit in implementing an N-way Join for the same latency and throughput as a 2-way Fork, for large N. All N signals must be synchronized at the NAND and negated-input AND gates at the start of the Forward and Backward Transfer parts. Beyond N=3, we might want to build trees of Join modules.

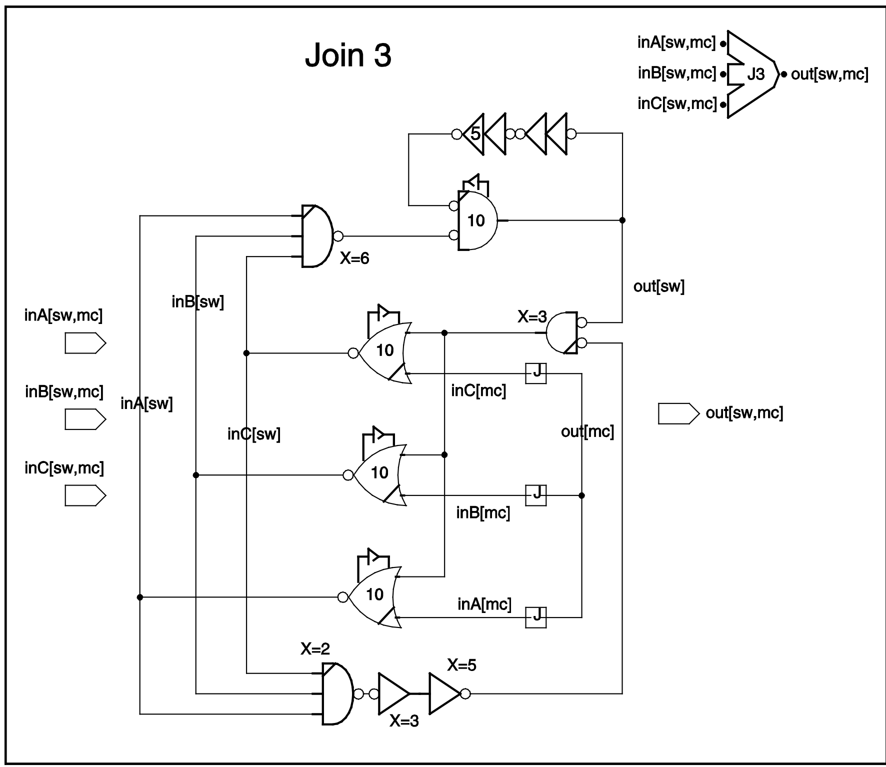


Figure 29: Circuit for Telescope GasP control module Join3, which is a 3-way Join. It has three input statewires with a master clear signal, **inA[sw,mc]**, **inB[sw,mc]**, and **inB[sw,mc]**, and one output statewire with a joined master clear, **out[sw,mc]**. We will use the icon in the top-right corner of this picture, when we represent a 3-way Join module. This design is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for inA[sw]-inB[sw] and out[sw].

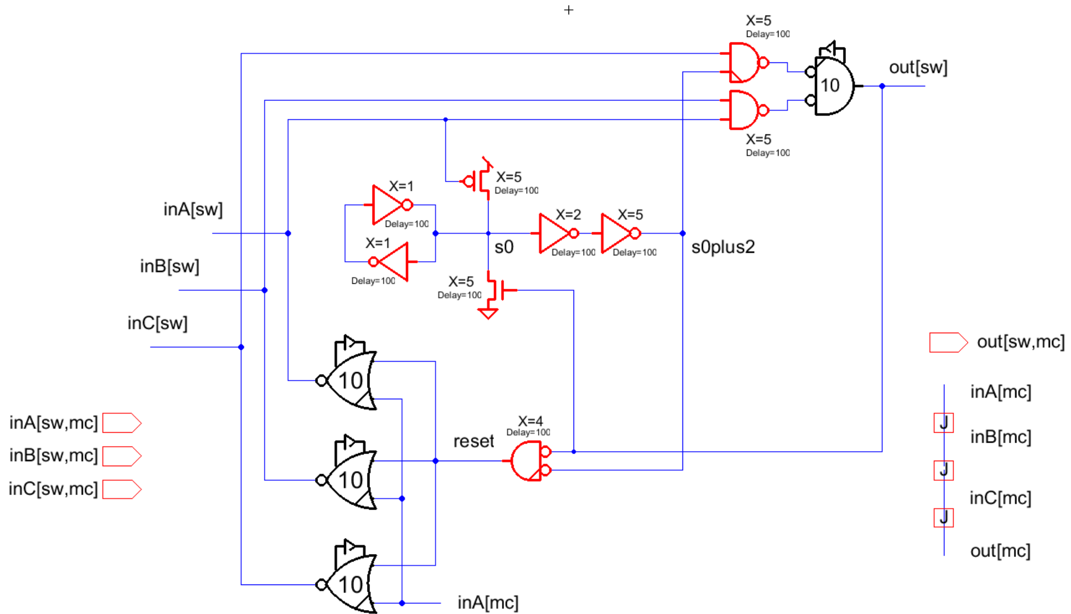


Figure 30: Second Telescope GasP implementation for module Join3. Here, we share the self-resetting loops between incoming and outgoing statewires insofar possible. This implementation is saved in the library name TelescopeGasP_wStateSharing, under module name Join3T (appendix `T' in the name stands for `Telescope').

7. Conclusion and Future Work

This document gives the implementation and simulation details of a GasP broadcast module, Store, that stores data, and of Telescope GasP implementations for other broadcast modules that do not store data but are used in ARCwelder [5,6]: Amplify, Fork, and Join.

Telescope GasP implementations have a minimum cumulative cycle time of 10+4k gate delays, where k is the number of consecutive Telescope GasP modules in the data path following and including the present module up the Store modules that store the computed results at the end of the data path. The increase in cycle time of 4 gate delays per data path module is due to the use of single-track channels.

This puts single-track designs styles, like Telescope GasP, at a disadvantage over other telescoping handshake design styles that use multi-track handshake signaling, such as Click [13]. We can alleviate this disadvantage and obtain a cumulative cycle time of 10+2k gate delays by adding a fast reset strategy.

The fast reset solutions presented in this document share two key features:

1. The fast reset signal is generated from the final Store module in the data path. We anticipate that, in case of the Fork module, we will be able to identify a particular Store module from which we can tap its fast reset signal. We leave such identification for future work.
2. We re-use the existing master clear input signal to the module as fast reset signal input. This was the easiest way to create a fast reset, without changing the module. For practical use, though, the master clear signal operates through an almost three times weaker NMOS transistor than the transistors and transistor stacks used in normal post-initialized Telescope GasP operations. This makes the master clear input too weak to use as a fast reset, as is visible in the SPICE simulation results for the Join2 module – see caption of **Figure 20** on page 22. In the future, we will use a separate fast reset input that operates according to the logical effort expectations of a normal post-initialized Telescope GasP operation.

Appendix B

Telescope GasP: Narrowcast

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapters 2 and 3 and can be found as reference [26] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland.
  Telescope GasP Narrowcast and Arbitrated or Data Driven Control: Merge, Branch, and Distribute. *Technical Report, ARC2012-smg03, Asynchronous Research Center, Portland State University*, March, 2012.

Asynchronous Research Center
Portland State University

Subject: **Telescope GasP Narrowcast and Arbitrated or Data Driven Control: Merge, Branch, and Distribute**
Date: **1 March, 2012**
From: **Swetha Mettala Gilla, Marly Roncken and Ivan Sutherland**
ARC#: **2012-smg03**

References:

[1] Peter Beerel, Georgios Dimou, and Andrew Lines. Proteus: an ASIC Flow for GHz Asynchronous Designs, *IEEE Design & Test of Computers, Vol. 28, Issue 5*, pp. 36-51, 2011.
[2] Andrew Bardsley, Luis Tarazona, and Doug Edwards. Teak: A Token-Flow Implementation for the Balsa Language, In *Proceedings of the International Conference on the Application of Concurrency to System Design (ACSD)*, pp. 23-31, 2009.
[3] Teak asynchronous synthesis, Univ. of Manchester: http://apt.cs.man.ac.uk/projects/teak/.
[4] Jo Ebergen, Bill Coates, and Austin Lee. Long-Distance On-Chip Communication using GasP. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 119-116, 2011.
[5] Willem Mallon and Ivan Sutherland. Icons for Click and GasP Modules, *ARC2011-is05, Technical Report, Asynchronous Research Center, Portland State University,* March 2011.
[6] Asynchronous Research Center website: http://arc.cecs.pdx.edu/.
[7] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Long-Range GasP with Charge Relaxation. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 185-195, 2010.
[8] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Telescope GasP: Overview. *ARC2012-smg01, Technical Report, Asynchronous Research Center, Portland State University,* 2012. **Note: ARC2012-smg01 has been fully integrated into Chapter 2 of Swetha's Ph.D. thesis.**
[9] — GasP Storage and Telescope GasP Broadcast: Store, Amplify, Fork, and Join. *ARC2012-smg02.* **Note: ARC2012-smg02 is included as Appendix A in Swetha's Ph.D. thesis.**
[10] — Telescope GasP Narrowcast and Arbitrated or Data Driven Control: Merge, Branch, and Distribute. *ARC2012-smg03.* **Note: ARC2012-smg03 is included as Appendix B in Swetha's Ph.D. thesis.**
[11] — Telescope GasP: Repeat. *ARC2012-smg04.* **Note: ARC2012-smg04 is included as Appendix C in Swetha's Ph.D. thesis.**
[12] — Arbiter Design Improvements for GasP. *Technical Report, ARC2012-smg05, Work in Progress.* **Note: ARC2012-smg05 is now finished and included as Appendix G in Swetha's Ph.D. thesis.**
[13] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An Implementation Style for Data-Driven Compilation. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 3-14, 2010.
[14] Steven M. Rubin. Using the ELECTRIC™ VLSI Design System, Version 8.11. *Static Free Software and Sun Microsystems, ISBN 0-9727514-3-2,* R.L. Ranch Press, 2010.
[15] Charles Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems, Carver Mead and Lynn Conway (Eds), Addison-Wesley*, pages 218-262, 1980.
[16] Ivan Sutherland. A Mutual Exclusion Pass Gate, *ARC2010-is26, Technical Report, Asynchronous Research Center, Portland State University,* May 2010.
[17] Ivan Sutherland. Fourth Class Handout: Proper Stopper, *ARC2012-is49, Technical Report Asynchronous Research Center, Portland State University,* October 2010.
[18] Ivan Sutherland and Scott Fairbanks. GasP: A Minimal FIFO Control. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 46–53, 2001.
[19] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow, In *Proc. International Workshop on Logic Synthesis, 2004.*

This document contains information developed at the Asynchronous Research Center at Portland State University. Disclose this information to whomever you choose. Distribution is permitted with adequate reference to the source of the ideas and information. You may reproduce this material for any educational use. Copies of the material must contain this notice.

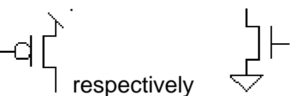
Abstract
This document gives Telescope GasP implementations for pipeline modules for narrowcast 1-to-1ofN or 1ofN-to-1, or for 1-to-MofN communications under arbitration or data control. These modules are used in ARCwelder [5,6]. ARCwelder, previously called TPDesign, is a design and compilation environment for dataflow computations; we use it to design self-timed systems. The organization of ARCwelder builds upon results and key learnings of the data-driven compiler effort by Handshake Solutions [13]. Related self-timed compilation efforts can be found in for instance [1,2,3,19].

An overall motivation for Telescope GasP, including pros, cons, and alternatives, can be found in [8]. Telescope GasP implementations for broadcast 1-to-NofN or NofN-to-1 follow in [9]. Telescope GasP implementations for Repeat modules follow in [11]. The references in the current ARC report list all references on Telescope GasP used here or in [8,9,10,11].

Keywords: GasP backend for ARCwelder (TPDesign), Telescope GasP for narrowcast and arbitrated or data-driven control.

Notation:

1. All GasP implementations in this document and in the related documents [8,9,10,11] are non-inverting, i.e., their handshake channels, a.k.a. statewires, use the same encoding. Statewires say whether the data bundled with the statewire are valid. We use:
   - HI for a high voltage level, e.g. VDD, to indicate a handshake REQUEST phase with valid data.
   - LO for a low voltage level, e.g. VSS or GND, to indicate a handshake ACKNOWLEDGE phase where data are no longer needed.
     In our circuit diagrams, we use a short 45-degree line segment for VDD and a triangle for VSS. A PMOS transistor connected to VDD and an NMOS transistor connected to VSS are depicted as:



2. Our schematic designs use a so-called JBOX construct. This is a special construct in Electric [14] to connect signals with different names. The symbol for a JBOX construct is a box with the letter J in it. We specifically use it to join the **master clear (mc)** signals on the various statewires of a module. For example, **Figure 11** on page 14 uses a JBOX to connect master clear signals in[mc] and out[mc].

3. It turns out that the telescoping handshake relation requires extra relative timing constraints that are not required for conventional GasP [18,7]. The constraints that we need to obtain a telescopic handshake relation tend to be transition specific. If the delay margins are small, e.g. in the order of 1 gate delay, we may want to add extra delay circuitry to increase the margin from say 1 to 3 gate delays. To delay a transition from **sin** LO to **sout** HI by two additional gate delays we will use the left-hand circuit in **Figure 1** below, and to delay a transition from **sin** HI to **sout** LO by two additional gate delays, we will use the right-hand circuit in **Figure 1**.

   The two delay circuits in **Figure 1** add two gate delays to the targeted **sin-**to-**sout** transition. Depending on the actual cycle time of the module in the given data path, we can extend this further to four additional gate delays or more, without losing throughput. These two types of delay circuits are sufficiently generic for our needs, and have the following properties:
   - The required transition is slowed down by sufficient margin.
   - The required transition is fast enough to sustain the maximum throughput.
   - The delay of the opposite transition matches the original delay without the delay circuit.

   When we present the circuit schematics of a telescope GasP component, we will indicate places with a 1 gate delay margin where one of the delay circuits in **Figure 1** may be needed.

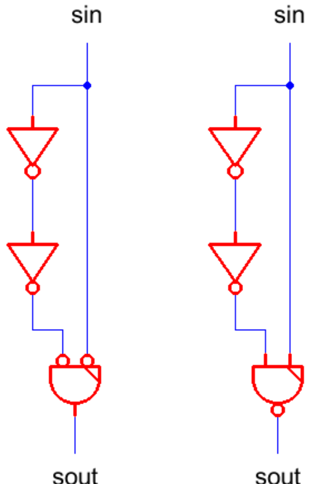


Figure 1: Delay circuits to increase the delay margin from **sin** LO to **sout** HI by two gate delays (left), and to increase the margin from **sin** HI to **sout** LO by two gate delays (right). Other path delays remain unchanged.

Acknowledgements: We gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD thesis work.

Table of Contents

References: 1
1. INTRODUCTION 4
2. Custom Driver Designs for Merge, Branch, and Distribute 6
   2.1. Single-Track Statewire Drivers for Pull-HI and Keep-LO 6
   2.2. Single-Track Statewire Drivers for Pull-LO and Keep-HI 8
3. Data Steering Logic for Merge, Branch, and Distribute 12
4. Merge Modules 13
   4.1. Default Merge Module, or Merge 13
   4.2. Arbitrated Merge Module 19
   4.3. 2-Way Sequential Merge Module, or Toggle Merge 28
   4.4. 3-Way Sequential Merge Module 33
5. Branch Modules 35
   5.1. 2-Way Sequential Branch Module, or Toggle Branch 35
   5.2. 3-Way Sequential Branch Module 41
   5.3. 4-Way Sequential Branch Module 43
6. Distribute Modules 45
   6.1. 2-Way Distribute Module 45
   6.2. 3-Way Distribute Module 51
   6.3. 4-Way Distribute Module 53
7. Conclusion and Future Work 55

1. INTRODUCTION

This document gives GasP control implementations for the following pipeline modules used in ARCwelder and specified and listed in [5,6]: Merge, Branch, and Distribute. These modules facilitate narrowcast communication for 1ofN-to-1 or 1-to-1ofN or 1-to-MofN modules. The selection can be sender-driven, arbitrated or data-driven. Below follows a short description of each module. We will focus on dataflow aspects and leave out handshake details. We grouped some of the modules to emphasize their similarity.

- **Merge and Branch**
  - Merge and Branch modules form a companion pair. They generalize the single input, single output streaming by selecting one of many inputs (Merge) or one of many outputs (Branch). We give Telescope GasP implementations for 2-way Merge and Branch modules.
  - We cover the full type set of Merge modules: (1) a default Merge whose inputs are mutually exclusive, (2) an arbitrated Merge that can handle concurrent inputs, and (3) a sequential Merge that takes its inputs in a round-robin fashion.
  - At present, ARCwelder supports only the sequential Branch that takes its outputs in a round-robin fashion; we give Telescope GasP designs for a two-, three-, and four-way sequential Branch.
  - Implementation details for the Merge module follow in Section 4 of this document.
  - Implementation details for the Branch module follow in Section 5 of this document.
- **Distribute**
  - The Distribute module streams one input to a subset of N-way outputs. The subset varies per communication and is set via N additional incoming control bits. We give Telescope GasP designs for 2-way, 3-way and 4-way Distribute modules.
  - Implementation details for the Distribute module follow in Section 6 of this document.

Note: The following text up to Section 2 has been copied from [9], to make this document self-contained.

When designing Telescope GasP modules, we distinguish and design four communication control parts:

1. **Forward Transfer**
   The HI handshake request signal coming in from the predecessor statewire transfers into a HI request signal on the successor statewire. The actual transfer generally requires synchronization with other incoming and with outgoing signals. We opted for a forward transfer latency of 6 gate delays for the Store module and of 2 gate delays for the other modules.
2. **Backward Transfer**
   The LO handshake acknowledge signal coming in from the successor statewire transfers into a LO acknowledge signal on the predecessor statewire. The transfer generally requires synchronization with other outgoing and with incoming signals. We opted for a backward transfer latency of 4 gate delays for the Store module and of 2 gate delays for the other modules.
3. **Forward Reset**
   The forwarded HI request signal on the successor statewire turns off its own HI drive after 5 gate delays, thereafter relying on the keeper to keep the statewire HI as long as needed. The Forward Reset in a storage-free Telescope GasP module strictly precedes its Backward Reset. Traditional GasP modules, like module Store, turn off their forward HI and backward LO drives concurrently.
4. **Backward Reset**
   In a storage-free Telescope GasP module, the backward transferred LO acknowledge signal on the predecessor statewire turns off its own LO drive after 5 gate delays. Traditional GasP modules, like module Store, turn off their forward HI and backward LO drives concurrently.

The two transfer parts each contain a driver and a half-keeper. Similar to the half-keeper design organizations of 4-2 Gasp in [18] and of 6-4 GasP in [7], the half-keeper serves two purposes: it prevents the statewire from floating when the drive ceases, and it avoids a keeper-driver short-current conflict.

We will explore two different approaches in designing Telescope GasP modules. The first approach uses separate self-resetting loops for incoming and outgoing handshake channels. The Merge module in **Figure 11** is an example of this first design approach. It has three self-resetting loops: the loop at the top is used to start and stop the HI drive for *out[sw],* while the two lower loops start and stop the LO drives for *inA[sw]* respectively *inB[sw]*. The second approach shares self-resetting loops between incoming and outgoing handshake channels where possible. The Merge module in the following **Figure 12** is an example of the latter design approach. It has two self-resetting loop to start and stop the HI-LO drives for *out[sw]-inA[sw]* respectively *out[sw]-inB[sw]*.

Telescope GasP implementations designed according to the first approach are saved under library name *TelescopeGasP_SeparateStates*. Implementations designed according to the second approach are saved under *TelescopeGasP_wStateSharing*.

Both approaches have their advantages and disadvantages. The advantage of the first approach used in **Figure 11** is that it supports more incoming and outgoing channels than the second approach used in **Figure 12**, because the first design approach is capable of spreading the input loads over multiple gates. This will become obvious when we discuss the sequential or round-robin Merge. The first approach easily handles a 3-input sequential Merge module, but the second approach does not.

The advantage of the second approach used in **Figure 12** is that it has larger delay margins than the first approach. The second approach has delay margins that we feel are safe, whereas the first approach has marginal relative timings that we may want to adjust. **Figure 11** has two such marginal relative timings:

1. After either *inA[sw]* or *inB[sw]* have risen, it takes 2 gate delays before *out[sw]* rises and disables the two inverted input AND gates in the two lower self-resetting loops, while it takes 3 gate delays for the risen incoming signal to enable its corresponding AND gate via its lower self-resetting loop. To guarantee a telescope relation between *inA[sw]* and *out[sw]* and between *inB[sw]* and *out[sw]*, the inverted input AND gate must sense the disabling *out[sw]* transition before it senses the enabling incoming transition. The delay margin between the two is only 1 gate delay.
2. After *out[sw]* falls, it takes 2 gate delays before the present HI signal on either *inA[sw]* or *inB[sw]* falls and disables the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the LO *out[sw]* signal to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *inA[sw]* and *out[sw]* and between *inB[sw]* and *out[sw],* the NAND gate must sense the disabling incoming transition before it senses the enabling *out[sw]* transition. The delay margin between the two is only1 gate delay.

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can make both margins as safe as the delay margins deployed in the second design approach, by adding extra circuitry to delay the transition that must arrive later by 2 more gate delays. Other transitions remain unaffected. For instance, to delay a transition from **sin** LO to **sout** HI we insert the left-hand circuit of **Figure 1**, and to delay a transition from **sin** HI to **sout** LO we insert the right-hand circuit of **Figure 1**.

One of the design criteria for the current Telescope GasP implementations is a low 2 gate delay latency between incoming and outgoing handshakes. From [8], we know that we need *at least* 2 gate delays to implement non-inverting Telescope GasP. Our design criterion is to make this latency *no more than* 2 gate delays. To be more precise: we want a minimum latency of 2 gate delays per module from the moment its last incoming handshake goes HI (starts) to the moment its outgoing handshakes go HI (start), and, likewise, a minimum latency of 2 gate delays from the moment its last outgoing handshake goes LO (resets) to the moment its incoming handshakes go LO (reset).

To guarantee this low latency, some of the control functionality is inevitably forced into the drivers for the handshake channels. This results in a myriad of custom driver designs, each with 1 gate delay latency. Section 2 below lists all custom design drivers used for the design of Merge, Branch, and Distribute modules, and includes their circuit schematics.

2. Custom Driver Designs for Merge, Branch, and Distribute

To achieve a small 2 gate delay transfer latency, some of the forward and backward transfer logic gets integrated inevitably into the driver and half-keeper logic. As a result, Telescope GasP modules use a large variation of driver and half-keeper logic for incoming and outgoing statewires - larger than we are used to see in traditional, e.g. 6-4, GasP modules.

Section 2.1 gives the implementations of the HI driver and LO half-keeper logic for successor drivers of outgoing statewires of narrowcast and arbitrated or data-driven control modules. Section 2.2 does the same for the LO driver and HI half-keeper logic for predecessor drivers of incoming statewires. By default, we assume that the initial value of a statewire is LO, and so, we will add a **master clear (mc)** signal and related circuitry for fast return-to-zero resetting to the LO driver logic.

2.1. Single-Track Statewire Drivers for Pull-HI and Keep-LO

Our Merge, Branch, and Distribute implementations use the following HI driver and LO half-keeper logic for successor drivers:

- **Succ Driver:** see [9].
- **Succ AND Driver:** see [9].
- **Succ OR Driver:** 2 LO inputs, either of which drives the statewire HI, and a half-keeper to keep it LO.
- **Succ OR3 Driver:** 3 LO inputs, either driving the statewire HI, and a half-keeper to keep it LO.
- **Succ ORAND Driver:** 3 LO inputs ORAND-ed to drive the statewire HI and half-keeper to keep it LO.
- **Succ OR2xAND Driver:** 4 LO inputs ORAND-ed to drive the statewire HI; half-keeper to keep it LO.

All five circuits have an input-to-output transition time of one gate delay. Implementations not already given in [9] follow in **Figure 2** to **Figure 5** below.

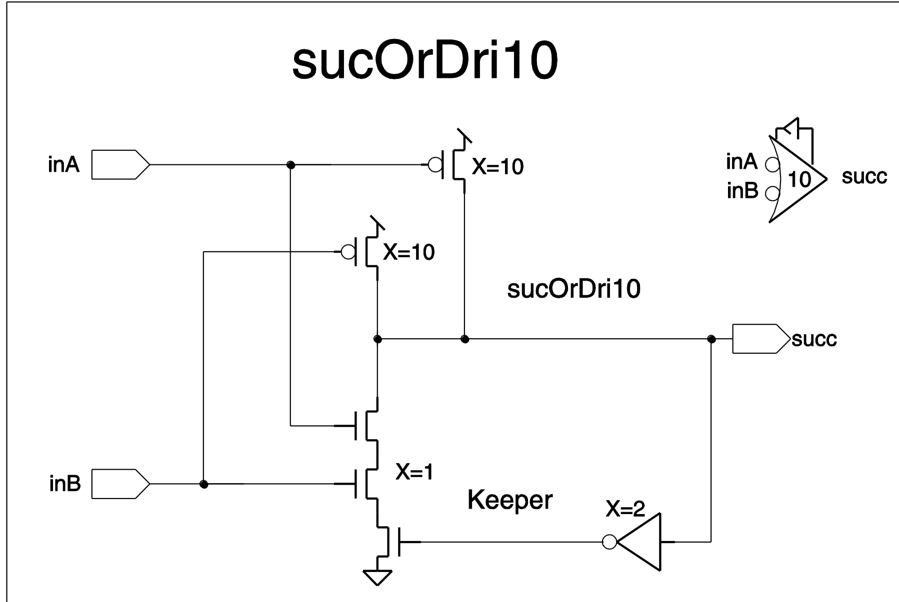


Figure 2: Transistor-level GasP design for **Succ OR Driver**. Either input **inA** or **inB** will drive statewire **succ** HI within one gate delay. The half-keeper is there to keep **succ** LO, when needed. The official 180nm cell name in our Electric design environment [14] is **SucOrDri10**. We use the icon in the top right corner as a gate-level shortcut for Succ OR Driver. It is used in the default Merge module.

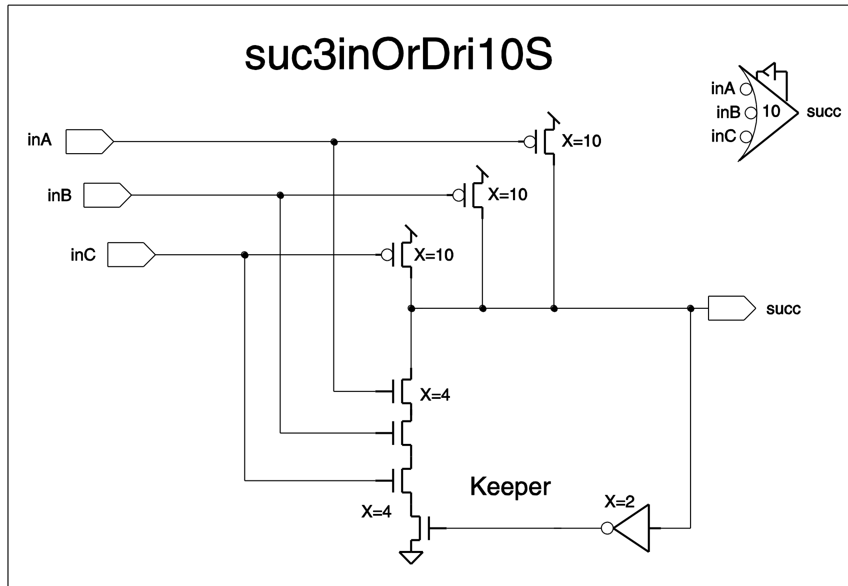


Figure 3: Transistor-level GasP design for **Succ OR3 Driver**, with three inputs, named **inA**, **inB** and **inC**, either of which will drive statewire **succ** HI within one gate delay, and a half-keeper to keep **succ** LO. This official cell name is **suc3inOrDri10S**. We will use the icon in the top right corner of the picture as a gate-level shortcut for this driver design. It is used in the three-way Sequential Merge module.

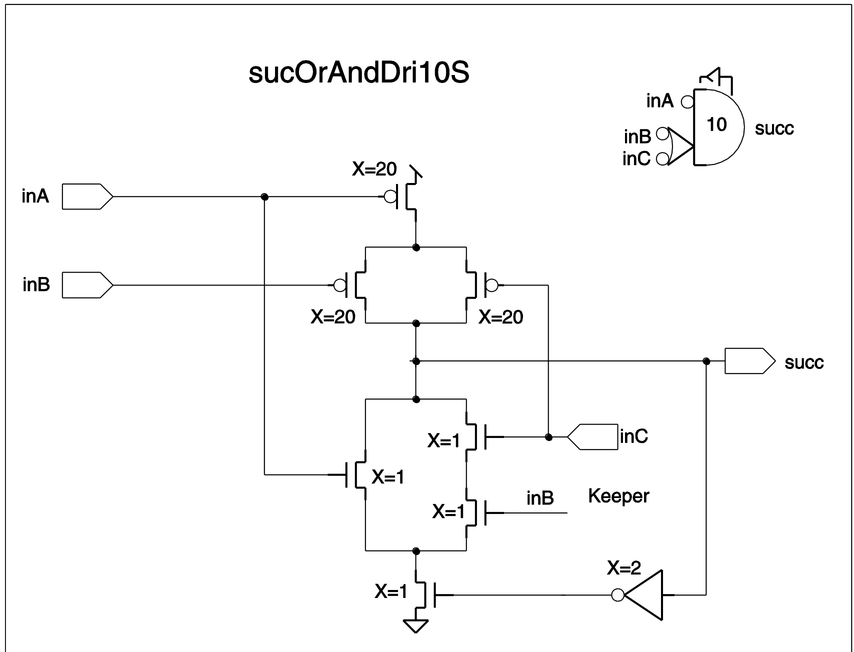


Figure 4: Transistor-level GasP design for **Succ ORAND Driver**. Its official cell name is **sucOrAndDri10S**. When **inA** is low and either **inB** or **inC** are low, statewire **succ** is driven HI within one gate delay. The half-keeper is there to keep **succ** LO, when needed. We will use the icon in the top right corner of the picture as a gate-level shortcut for this driver design. It is used in our first Arbitrated Merge design.

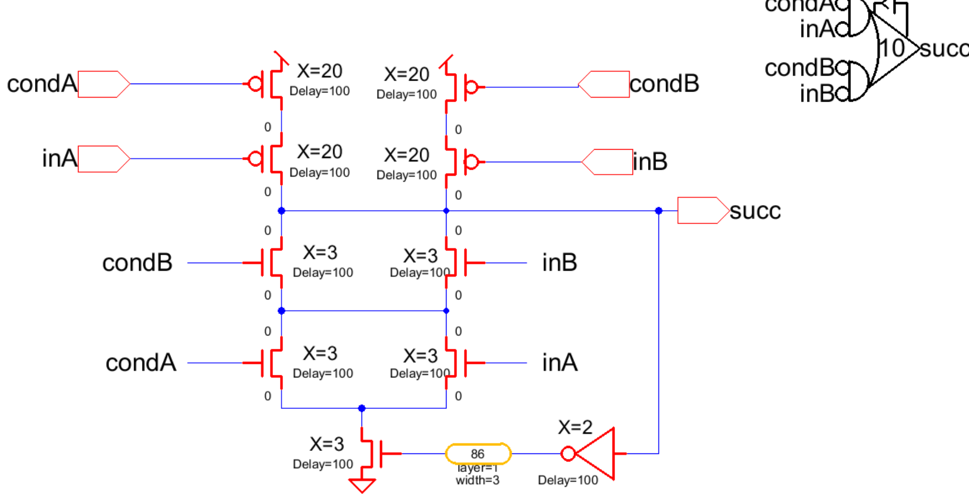


Figure 5: Transistor-level GasP design **Succ OR2xAND Driver**. Its official cell name is **sucOr2xAndDri10**. When either **condA** and **inA** are both low or **condB** and **inB** are both low, statewire **succ** is driven HI within one gate delay. The half-keeper is there to keep **succ** LO, when needed. We use the icon in the top-right corner as a gate-level shortcut for this driver design. It is used in our second Arbitrated Merge design, and saved under the library for TelescopeGasP_wStateSharing

2.2. Single-Track Statewire Drivers for Pull-LO and Keep-HI

Our Merge, Branch, and Distribute implementations use the following HI driver and LO half-keeper logic for predecessor drivers:

- **Pred Driver:** see [9].
- **Pred AND Driver:** see [9].
- **Pred OR Driver:** 2 HI inputs, either of which drives the statewire LO, and a half-keeper to keep it HI.
- **Pred OR3 Driver:** 3 HI inputs, either of which drives the statewire LO, and a half-keeper to keep it HI.
- **Pred Driver with MC and fastRTZ:** Pred Driver with extra 2-input guarded fastRTZ facility.
- **Pred OR Driver with MC and fastRTZ:** Pred OR Driver with extra 2-input guarded fastRTZ facility.
- **Pred AND Driver with MC and fastRTZ:** Pred AND Driver with extra 2-input guarded fastRTZ facility.

All seven circuits have an input-to-output transition time of one gate delay. By convention, statewires are initialized LO. So, unless otherwise specified, LO drivers have a **master clear (mc)** input signal and related circuitry to initialize the statewire to LO. All driver implementations in this Section follow this convention. Also any additional circuitry for fast resetting is integrated into the LO driver logic for predecessor drivers of incoming statewires.

GasP implementations not already given in [9] follow in **Figure 6** to **Figure 10** below.

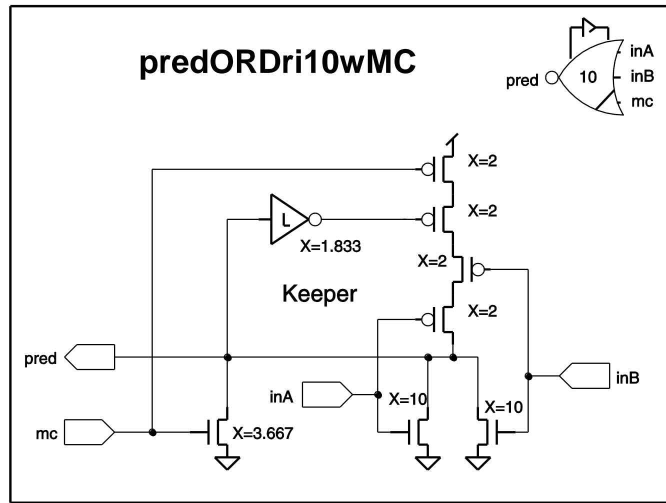


Figure 6: Transistor-level GasP design for **Pred OR Driver**, with two inputs **inA** and **inB**, either of which will drive statewire **pred** LO within one gate delay. The official 180nm cell name in our Electric design environment is **predOrDri10wMC**. The half-keeper is there to keep **pred** HI, when needed. Input signal **mc**, for **master clear**, is used to initialize pred to LO. We will use the top-right as a gate-level shortcut for this design. It is used in the two-way Sequential Branch module, also known as Toggle Branch or two-way Round-Robin Branch.

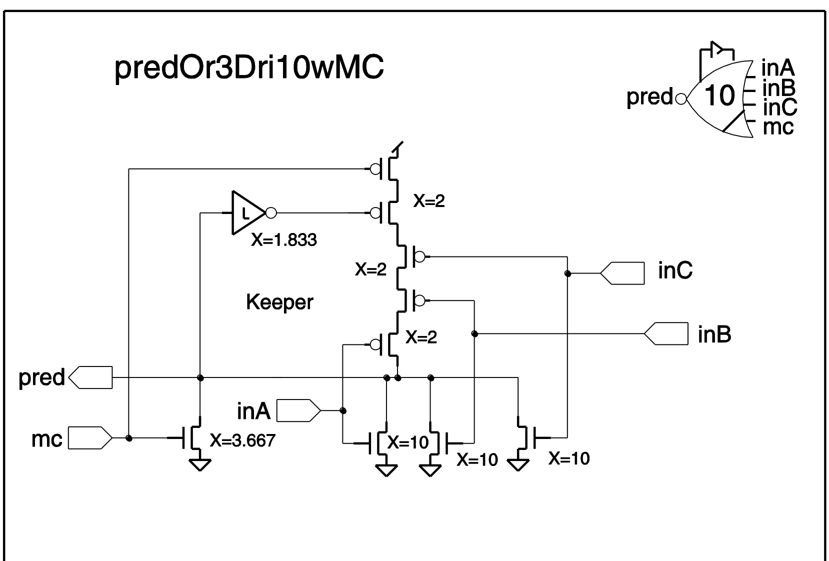


Figure 7: Transistor-level GasP design for **Pred OR3 Driver**, with three inputs **inA, inB, inC,** either of which will drive statewire **pred** LO within one gate delay. Its official name is **predOr3Dri10wMC**. The half-keeper is there to keep **pred** HI, when needed. Input signal **mc**, for **master clear**, is used to initialize **pred** to LO. We will use the icon in the top right corner of the picture as a gate-level shortcut for this driver design. It is used in the three-way Sequential Branch module.

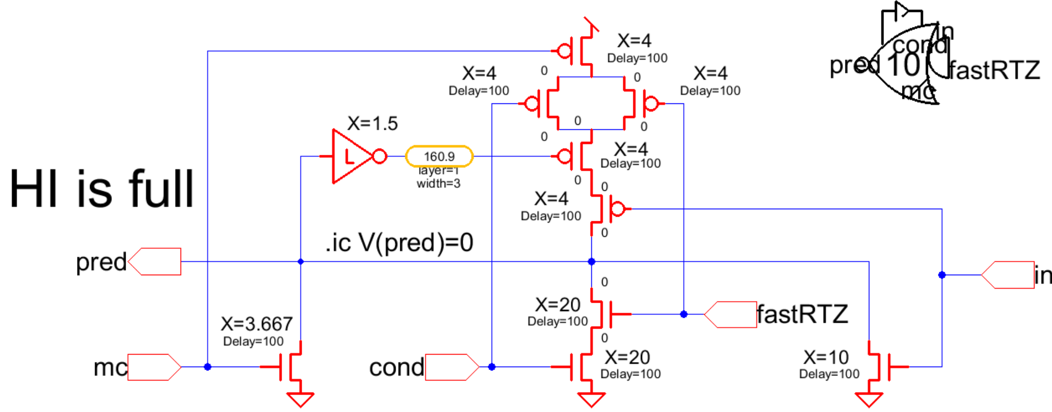


Figure 8: Transistor-level GasP design for predecessor driver **Pred Driver with MC and fastRTZ**. Its official name is **predDri10wMCwFastRTZ**. For normal operations, this driver acts like a Pred Driver with input signal **inA** to drive statewire **pred** LO within one gate delay, and with master clear signal **mc** to initialize **pred** to LO. We added a conditional fast reset operation: when both **cond** and **fastRTZ** are HI input stat wire **pred** is driven LO within one gate delay. The half-keeper is there to keep **pred** HI, when needed. We will use the icon in the top-right corner of the picture as a gate-level shortcut for this driver design. It is used in our second design for the two-way Sequential Merge module, also known as Toggle Merge or two-way Round-Robin Merge. This predecessor driver is saved under the library for TelescopeGasP_wStateSharing.

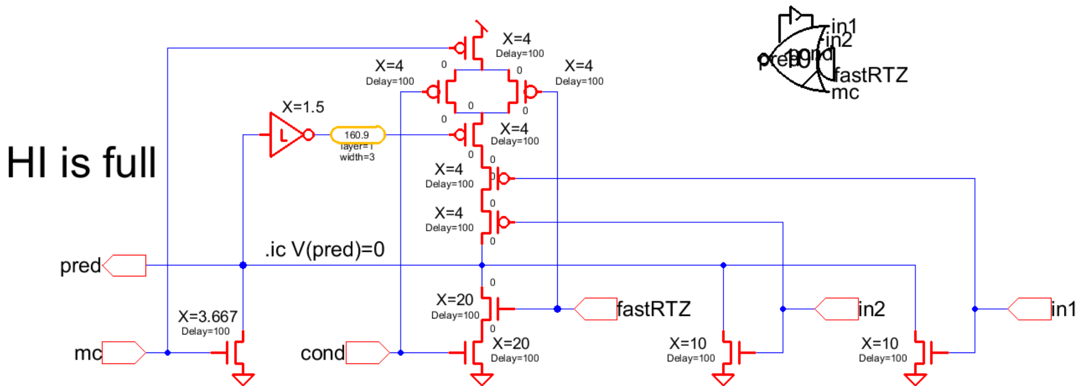


Figure 9: Transistor-level GasP design for predecessor driver **Pred OR Driver with MC and fastRTZ**. Its official name is **predORDri10wMCwFastRTZ**. For normal operations, this driver acts like a Pred OR Driver with either input signal **in1** or input signal **in2** driving statewire **pred** LO within one gate delay, and with master clear signal **mc** to initialize **pred** to LO. We added a conditional fast reset operation: when both **cond** and **fastRTZ** are HI input stat wire **pred** is driven LO within one gate delay. The half-keeper is there to keep **pred** HI, when needed. We will use the top-right icon as a gate-level shortcut for this driver design. It is used in our second design for the two-way Sequential Branch module, also known as Toggle Branch or two-way Round-Robin Branch. This predecessor driver is saved under the library for TelescopeGasP_wStateSharing.

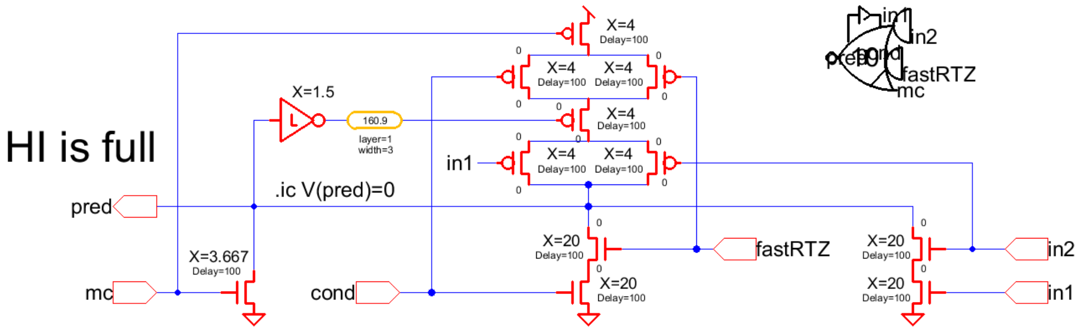


Figure 10: Transistor-level GasP design for **Pred AND Driver with MC and fastRTZ**. Its official name is **predANDDri10wMCwFastRTZ**. For normal operations, this driver acts like a Pred AND Driver with two HI input signals **in1** and **in2** to jointly drive statewire **pred** LO within one gate delay, and with master clear signal **mc** to initialize **pred** to LO. We added a conditional fast reset operation: when both **cond** and **fastRTZ** are HI input stat wire **pred** is driven LO within one gate delay. The half-keeper is there to keep **pred** HI, when needed. We will use the top-right icon as a gate-level shortcut for this driver design. It is used in our second design for the two-way Distribute module, and it is saved under the library for TelescopeGasP_wStateSharing.

3. Data Steering Logic for Merge, Branch, and Distribute

Unlike the broadcast modules in [9], the modules in this document steer data selectively: some data are forwarded to the outgoing communication channels, other data are ignored. We forward only those data bits that are bundled with incoming handshake channels that participate in the current telescope action.

We opted to broadcast the incoming data to all outgoing handshake channels, no matter whether or not the receiving module participates in the action. As a result, no special steering logic is needed for the single-input, multiple-output Branch and Distribute modules.

That leaves us with the Merge modules. Merge modules *do* need special steering logic. For each Merge module, we will design a special data steering signal per incoming handshake channel. The steering signal will be HI when the data bundled with the handshake channel are to be forwarded, and it will be LO when the bundled data are to be ignored. Such steering signals can be used to control the multiplexers in the data path that multiplex the data bits coming into the module onto the module's outgoing data bits. Our Telescope GasP designs focus primarily on the handshake circuitry. As such, our design implementations of the Merge modules will show the data steering signals, but not the bit-wise multiplexers in the data path.

4. Merge Modules

Merge modules generalize the single input, single output streaming of an Amplify module by selecting one of many inputs. In short, Merge modules provide 1ofN-to-1 communication. In Sections 4.1 to 4.4 below, we give Telescope GasP implementations for a two-way un-arbitrated Merge, a two-way arbitrated Merge, and two- to four-way sequential Merge modules.

4.1. Default Merge Module, or Merge

The Default Merge module, which we will simply call Merge, provides single input to single output communication. Upon receiving exactly one out of two possible input communication requests, it will perform a corresponding output communication, using the data of the selected input source as output data. The Merge module assumes that the environment will send it input communication requests in a mutually exclusive fashion. We can tap the data steering signals from the control inputs, because these remain stable until the end of the output communication.

Figure 11 and **Figure 12** show our Telescope GasP circuit implementation for the Merge module. The implementation in **Figure 11** is an example of our first design approach with independent self-resetting loops for statewire of incoming and outgoing handshake channels. It has three self-resetting loops: the loop at the top is used to start and stop the HI drive for *out[sw],* the two loop at the bottom are used to start and stop the LO drives for *inA[sw]* respectively *inB[sw].* The Merge implementation in **Figure 12** is an example of the second design approach with shared self-resetting loops between the statewires of incoming and outgoing handshake channels. It has two self-resetting loops, with separate taps to start and stop the HI drive for *out[sw]* and the LO drives for *inA[sw]* and *inB[sw].*

A SPICE-level simulation of the handshake behavior of the Merge implementation in **Figure 11** follows in **Figure 13**. The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **inA[sw]**, **inB[sw]**, **out[sw]**, and **mc** are LO. When either **inA[sw]** or **inB[sw]** go HI, indicating mutual exclusive presence of data on the incoming channels, and when **out[sw]** is LO, indicating the availability of space, the following four sets of actions are started **in sequence**, as follows:

1. The Forward Transfer part signals the presence of new data to the successor module by driving **out[sw]** HI. This takes 2 gate delays. The steering signal for the HI incoming channel is driven HI.
2. The Forward Reset section kicks in, cutting off the HI forward drive to a 5 gate delay pulse signal.
3. The Backward Transfer waits until **out[sw]** is LO, and then reports new availability of space to the predecessor by driving the presently HI **inA[sw]** respectively **inB[sw]** LO, 2 gate delays later.
4. The Backward Reset part kicks in, cutting off both LO backward drives after 5 gate delays.

After action 4, the Merge waits until again either **inA[sw]** or **inB[sw]** is HI, so it can start its next cycle.

Internally, the behaviors of the two Merge implementations differ. The advantage of the implementation in **Figure 12** is that it has delay margins that are relatively safe, whereas the implementation in **Figure 11** has two delay margins that are marginal. The differences in delay margin are very similar to those described in [9] for the two implementations of the broadcast modules. The two marginal delay margins in **Figure 11** are as follows:

1. After either *inA[sw]* or *inB[sw]* have risen, it takes 2 gate delays before *out[sw]* rises and disables the two inverted input AND gates in the two lower self-resetting loops, while it takes 3 gate delays for the risen incoming signal to enable its corresponding AND gate via its lower self-resetting loop. To guarantee a telescope relation between *inA[sw]* and *out[sw]* and between *inB[sw]* and *out[sw]*, the inverted input AND gate must sense the disabling *out[sw]* transition before it senses the enabling incoming transition. The delay margin between the two is only 1 gate delay.
2. After *out[sw]* falls, it takes 2 gate delays before the present HI signal on either *inA[sw]* or *inB[sw]* falls and disables the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the LO *out[sw]* signal to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *inA[sw]* and *out[sw]* and between *inB[sw]* and *out[sw],* the NAND gate must sense the disabling incoming transition before it senses the enabling *out[sw]* transition. The delay margin between the two is only1 gate delay.

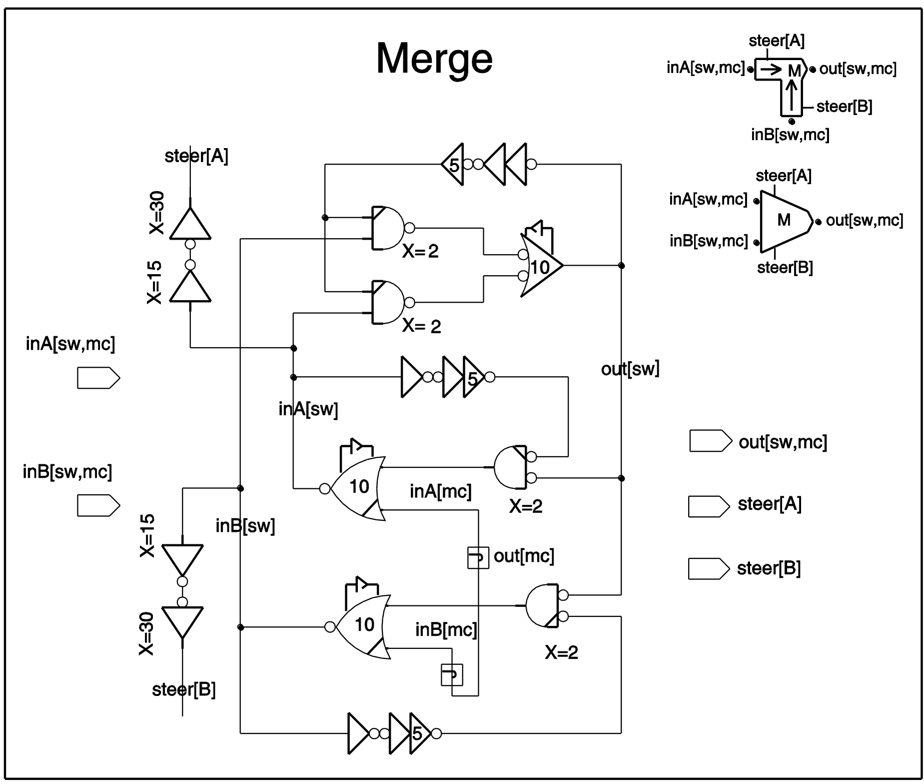


Figure 11: Telescope GasP implementation of module Merge. The circuit has two input statewires with a master clear signal, **inA[sw,mc]** and **inB[sw,mc]**, two corresponding data steering signals **steer[A]** and **steer[B]**, and one output statewire with a forwarded master clear, **out[sw,mc]**. We will use either of the two icons in the top-right corner of this picture, when we represent a Default Merge module. This particular implementation is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for **inA[sw]**, **inB[sw]**, and **out[sw].**

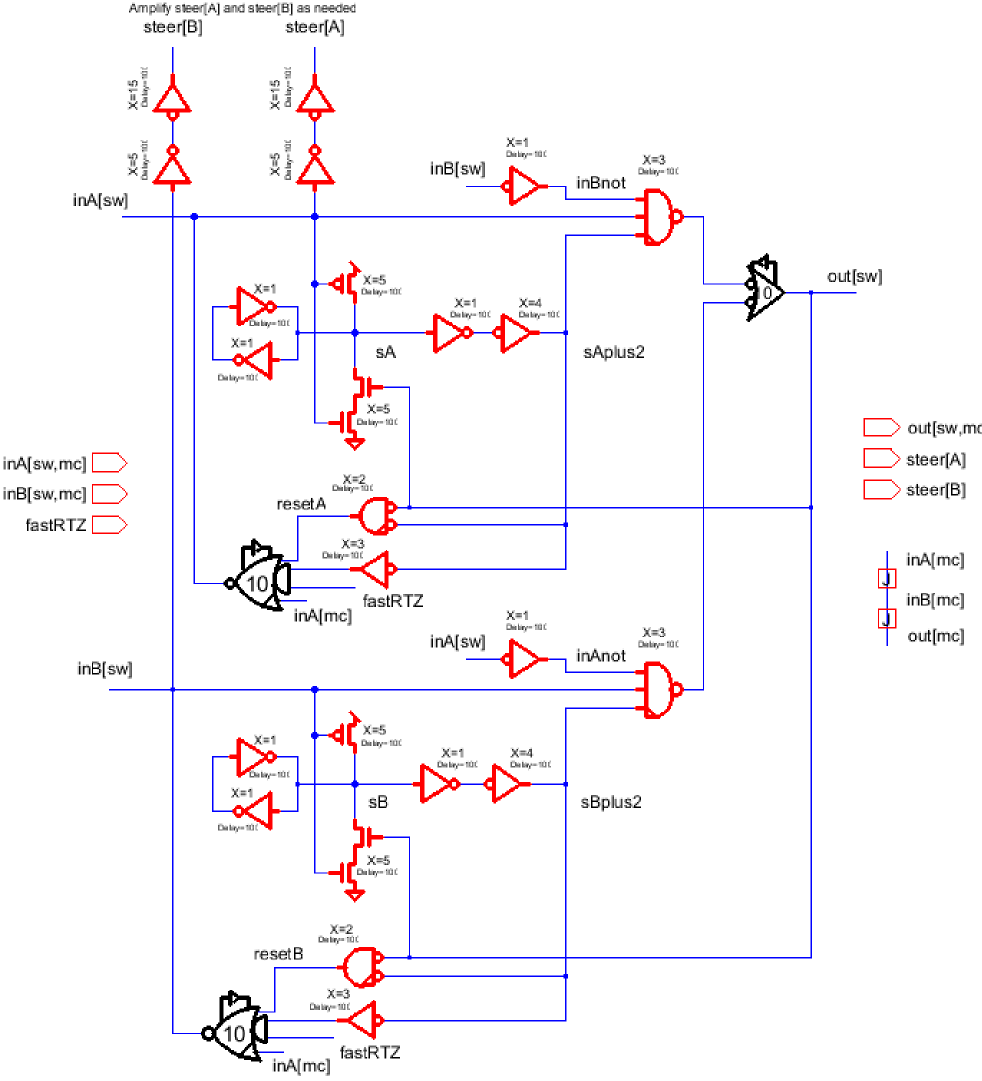


Figure 12: Second Telescope GasP implementation for module Merge. Here, we share the self-resetting loops between incoming and outgoing statewires insofar possible. This implementation is saved in the library named TelescopeGasP_wStateSharing, under module name Merge2TfastRTZ. Appendix `TfastRTZ' in the name indicates that this is a Telescope GasP implementation with fast reset capability. To ensure that we reset only the incoming handshake that caused **out[sw]** to go HI, we have AND-ed the fast reset signal at the LO driver with an internal guard signal. The guard for **inA[sw]** is tapped from internal state signal **sA** via three inversions, The guard for **inB[sw]** is tapped from internal state signal **sB** via three inversions.

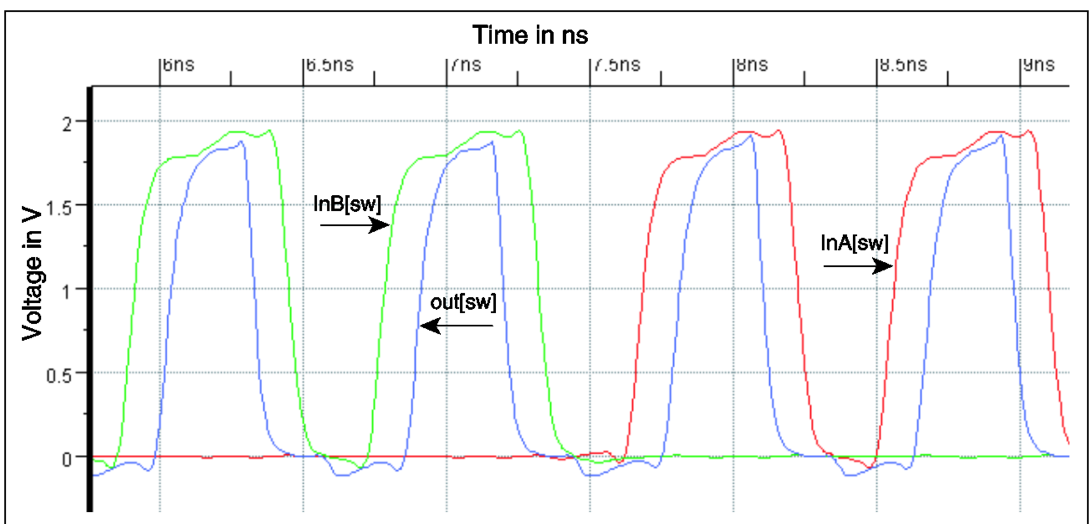


Figure 13: SPICE-level simulation of the Telescope GasP implementation of the Merge module in **Figure 11**. Red and green signals inA[sw] and inB[sw] have a telescopic handshake relationship to blue signal out[sw]: inA[sw] or inB[sw] HI causes out[sw] to go HI, and when out[sw] has gone LO, only then will the input go LO. The simulation clearly shows that each HI out[sw] pulse is contained within a HI pulse on inA[sw] or inB[sw].

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Merge module in **Figure 11** as safe as those in **Figure 12**.

The two implementations have the same cycle time and latencies. In both implementations, the Forward Transfer part is implemented using the Succ OR Driver of **Figure 2**. There is 1 additional gate between **inA[sw]** respectively **inB[sw]** and the input of the Succ OR Driver, giving a total forward latency of 2 gate delays. In both implementations, the two Backward Transfer parts are implemented using a Pred OR Driver. The Pred OR Drive version used in **Figure 12** has an additional fast reset, as shown in **Figure 9**. There is 1 additional gate between the outgoing channel **out[sw]** of each Merge implementation and the input of the two Pred OR Drivers, giving a total backward latency of 2 gate delays.

Without a fast reset signal, the minimum cycle time for each implementation is 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops in the circuit diagrams of **Figure 11** and **Figure 12**, assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each successor module between the Merge module and the following Store module. This skewing of the cycle time happens because each next module's Backward Transfer section must wait until the corresponding Forward Transfer in the module has completed.

Using a fast reset signal, we can reduce the backward latency and lower the cycle time to 12 gate delays per Merge module, with an increase in steps of 2 gate delays for each successor module between the Merge module and the following Store module. We designed a test environment that generates such a fast reset signal. We tested the behavior of the Merge module with this fast reset, using the test configuration in **Figure 14**. Unlike the fast reset solution that we presented in [9] for use in broadcast communication modules, in **Figure 14** we avoid sharing the master clear circuitry for fast resets. We have two reasons for providing a fast reset signal that is separate from the master clear signal:

- The first reason is due to the fact that all module implementations for Telescope GasP distribute the master clear signal over all incoming and outgoing channels, via a so-called JBOX construct, as explained in the beginning of this document. This makes it impossible to reset some inputs but not others. Though this did not matter for the broadcast modules in [9], it does matter for the narrowcast modules described in the current document.
- The second reason is that we prefer the fast reset drive to be stronger than the master clear signal drive. The master clear signal uses a weak 3.667 NMOS transistor to drive the predecessor statewire LO, as can be seen for instance in the Pred OR Driver in **Figure 6**. All other input signals to the Pred OR Driver use an NMOS transistor drive capability of 10 to drive the predecessor statewire LO. We would like the fast reset to have the stronger drive capability. **Figure 9** shows an implementation of the Pred OR Driver with such a stronger-drive fast reset.

Figure 14 shows a mutually exclusive sender environment with two senders on the left feeding a three-level deep tree of Merge modules. The Merge modules are designed in Telescope GasP using the implementation in **Figure 12**. The right-hand side shows the receiving half of a Store module [9]. We deleted the sending half from the Store implementation, because we don’t need it, and we added extra circuitry to generate a fast reset signal, which is called *mc_fastRTZ,* and to reset the outgoing handshake at the end of the Merge tree. This new configuration works in a way similar to the fast reset configurations for the broadcast modules in [9].

Four gate delays after the last incoming handshake to the Store module, i.e., after out[sw] has gone HI, the Store module makes mc_fastRTZ HI. At the same time, i.e., four gate delays after out[sw] has gone HI, the Store module makes its local clock signal *cl* HI. The mc_fastRTZ signal is distributed to each Merge module, as a separate input signal. The *cl* signal, besides clocking latches in the data path, also acts as input to the LO driver for out[sw]. Consequently, five gate delays after out[sw] has gone HI, all outstanding statewires that actively participate in the current telescoping communication are reset concurrently. The reset stops after 5 gate delays, via the self-resetting loop in the Store module. **Figure 15** and **Figure 16** show the corresponding SPICE-level simulation waveforms.

Specifically, **Figure 15** shows that *statewires that play no role* in the current telescoping communication *are not reset*. As an example, observe how statewire inB[sw] rises as soon as statewire inA[sw] has fallen, about midway through the top simulation and right in the middle of the first HI pulse on fastRTZ. The reverse happens during the second HI pulse on fastRTZ: as soon as statewire inB[sw] has fallen, statewire inA[sw] rises unhindered about midway during the fastRTZ pulse. The fact that only those handshakes are reset that actually participate in the present handshake telescope leading to out[sw] going HI, is due to the pairing of the fast reset signal with an internal guard signal. The guard signal indicates whether or not the incoming handshake plays a role in the current handshake telescope. See **Figure 12** for design details.

Merge2TfastRTZ_test_fastRTZ

smg-mr 7 Oct 2011

Figure 14: : Fast reset configuration and test environment to concurrently reset all Merge-connected statewires that actively participate in the current handshake telescope leading to out[sw] going HI. Note that, unlike the fast reset signal **mc_fastRTZ** in[9], the fast reset signal **fastRTZ** generated by the above Store module is independent of master clear signal **mc**.

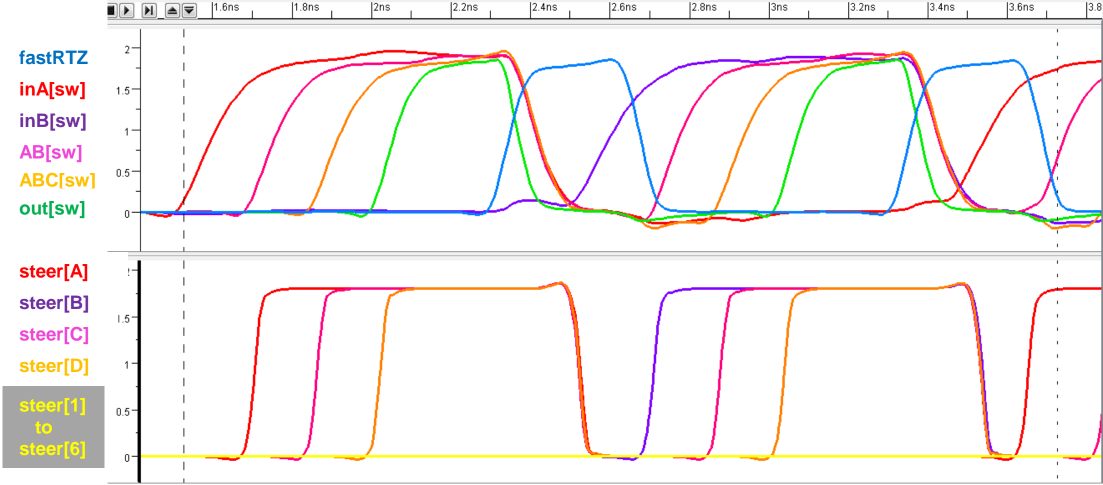


Figure 15: SPICE-level simulation results for the fast reset and test configuration in **Figure 14**.

- The top window shows how the blue HI pulse on fast reset signal fastRTZ concurrently resets all handshake signals that caused out[sw] to rise. The first HI pulse on fastRTZ has no effect on the rising purple inB[sw] signal, which is exactly how it should be as inB[sw] did not participate in the first out[sw] handshake. Likewise, the second HI pulse on fastRTZ has no effect on the rising red inA[sw] signal, which is also how it should be given that inA[sw] did not participate in the second out[sw] handshake. Signal out[sw] falls earlier than the other handshake signals. An explanation follows in **Figure 16** below.
- The bottom window shows the behaviors of the data steering signals for each Merge module. As expected, steer[A] rises 2 gate delays after inA[sw] rises, steer[C] two gate delays after AB[sw], and steer[D] two gate delays after ABC[sw]. The fall simultaneously during the fast reset pulse on fastRTZ. Note that steer[B] does not rise until the end of the fastRTZ pulse, after inB[sw] has risen. The yellow steering signals never rise: they represent the steering signals of statewires m1[sw] to m6[sw] unused by the given test setup. Note that the rising and falling slopes of the used steering signals are very steep. This is because they carry no load: the test configuration does not steer any data latches.

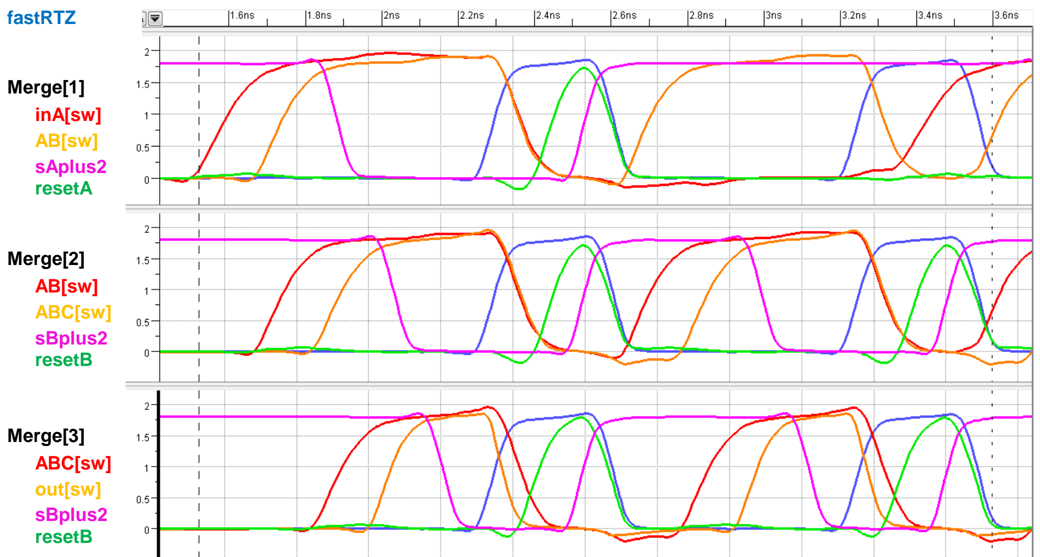


Figure 16: SPICE-level simulation details for the fast reset and test configuration in **Figure 14**. The top window contains waveforms for signals in the bottom-left Merge module, Merge[1].The middle and bottom windows show the waveforms for similar signals in the bottom-middle and rightmost modules of the tree, Merge[2] and Merge[3]. The waveforms display the slope differences between the falling handshake transitions on the module's output channel, in orange, and the module's input channel, in red. The falling slopes are almost identical for Merge[1] and Merge[2]. Only for Merge[3] do we see a difference. Below, we explain why this is so:

- The blue signal, fastRTZ, is the fast reset signal generated by the Store module. During its 5 gate delay HI pulse, all orange and red handshake signals are reset to LO, simultaneously.
- To understand why the falling slope for the outgoing handshake on out[sw] starts earlier and is slightly steeper than for the incoming handshake on ABC[sw] for Merge[3], AB[sw] for Merge[2], and inA[sw] for Merge[1], we must take a closer look at the LO drivers for the two statewires and at the LO driver input signals. Statewire out[sw] is connected to a Pred Driver in the successor Store module and driven by the local clock signal *cl*. Signal *cl* is amplified for the purpose of steering data latches, but the test configuration does not include these data latches. As a result, the load on signal cl is much smaller than the load on the fast reset signal, fastRTZ, though both are amplified in the same way. This causes *cl* to rise faster than fastRTZ, which in turn causes out[sw] to fall earlier than ABC[sw], AB[sw], and inA[sw].
- The green signal, resetA or resetB, is part of the normal backward reset path. When HI, it resets the red incoming handshake signal of the Merge module to LO, with or without the extra help of the guarded fastRTZ signal. From the Merge schematics in **Figure 12**, we can see that the guards for the internal reset and the fast reset originate from the same signal. Signal sAplus2 is used to reset inA[sw]. Signal sBplus2 is used to reset inB[sw]. If the guard holds, i.e., is HI, then the internal reset rises 1 gate delay after a falling transition on the orange outgoing handshake, and 2 gate delays after the blue fast reset rises.
- Unlike the fast reset signal that we generated for the broadcast modules in [9], which used a 3.667 strong NMOS transistor in the driver, the guarded fast reset for the Merge module uses an NMOS transistor stack of strength 10. Consequently, the red incoming handshake signals for the Merge modules have steeper reset slopes than the broadcast modules in [9]. As a result, the red signals are largely reset by the time the green internal reset starts, making the internal reset less effective during fast resets of Merge.
- Pink signal, sAplus2 or sBplus2, acts as (negated) guard for fastRTZ and as (confirming) guard for the internal reset. It is LO at the start of the fastRTZ HI pulse. It goes HI 4 gate delays after fastRTZ goes HI, and 1 more gate delay later, it stops the HI pulse on the green internal reset signal and its stops the fastRTZ drive to the red incoming handshake, thus ceasing the LO drive on the red incoming handshake. The 5 gate delay HI pulse on fastRTZ stops around the same time.

4.2. Arbitrated Merge Module

The Arbitrated Merge module provides single input to single output communication. Upon receiving one or more out of two possible input communication requests, it will arbitrate between the requests and non-deterministically select one to work with. For the selected incoming request, it performs a corresponding output communication using the data of the selected input data as output data. We can tap the data steering signals from the arbiter outputs, which remain stable until the output communication is complete.

Figure 17 and **Figure 18** show our Telescope GasP implementation for the Arbitrated Merge module. The implementation in **Figure 17** is an example of our first design approach with independent self-resetting loops for statewires of incoming and outgoing handshake channels. It has three self-resetting loops: the loop at the top is used to start and stop the HI drive for *out[sw],* the two loop at the bottom are used to start and stop the LO drives for *inA[sw]* respectively *inB[sw].* The Merge implementation in **Figure 18** is an example of the second design approach with shared self-resetting loops between the statewires of incoming and outgoing handshake channels. It has two self-resetting loops, with separate taps to start and stop the HI drive for *out[sw]* and the LO drives for *inA[sw]* and *inB[sw].*

A SPICE-level simulation of the handshake behavior of the Arbitrated Merge implementation in **Figure 17** follows in **Figure 21**(top). The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **inA[sw]**, **inB[sw]**, **out[sw]**, and **mc** are LO. When either or both of **inA[sw]** and **inB[sw]** go HI, indicating the presence of data on the incoming channels, and when **out[sw]** is LO, indicating the availability of space, the following four sets of actions are started **in sequence**, as follows:

1. The Forward Transfer part signals the presence of new data to the successor module by driving **out[sw]** HI. This takes 2 gate delays. Meanwhile, also the steering signal for the arbiter-selected incoming channel is driven HI.
2. The Forward Reset section kicks in, cutting off the HI forward drive to a 5 gate delay pulse signal.
3. The Backward Transfer waits until **out[sw]** is LO, and then reports new availability of space to the predecessor by driving the arbiter-selected incoming channel LO, 2 gate delays after observing the LO **out[sw]** signal. The forwarded data signals are reset LO.
4. The Backward Reset section kicks in, cutting off the LO backward drive to a 5 gate delay pulse.

At the end of these four **serial** sets of actions, the Arbitrated Merge module waits until either **inA[sw]** or **inB[sw]** is HI again, so it can start its next cycle.

Figure 19 and **Figure 20** show the circuit details of our arbiter design. The design is based on the **interlock element** in [Figure 7.25, 15] by Charles Seitz. We discussed this design in earlier ARC reports [16,17]. The organization in **Figure 19** reflects the layout organization of the arbiter in GasP. This layout has been proven topologically equivalent to the design by Seitz, using Electric [14].

We would like to emphasize that the arbiter is fair in that it will alternately select **inA[sw]** followed by **inB[sw]**, or vice versa, whenever these two inputs overlap. This is because it will take at least 5 gate delays for the selected input statewire to bounce back from LO to HI, while the unselected input statewire is already HI.

Figure 21(top) shows a SPICE-level simulation of the Arbitrated Merge module supporting the external handshake behavior of the arbitrated Merge module, described above. **Figure 21**(bottom) shows a SPICE simulated arbitrated resolution of two competing arbiter inputs and the corresponding changes in voltage levels for intermediate and output signals of the arbiter.

We are not quite happy with the voltage drop in the arbiter selected input – see **Figure 21**(top). We can diminish this voltage drop at the cost of a somewhat slower minimum latency through the arbiter. Swetha developed a new arbiter design with a sufficiently small voltage drop for the arbiter selected input. We use it in the second Arbitrated Merge design **Figure 18**. The resulting waveforms are visible in **Figure 23** and

Figure 24, and show relatively stable and well-defined voltage levels for the arbiter inputs. Swetha will write a separate ARC report with an analysis of the issues and solutions in the arbiter design [12].

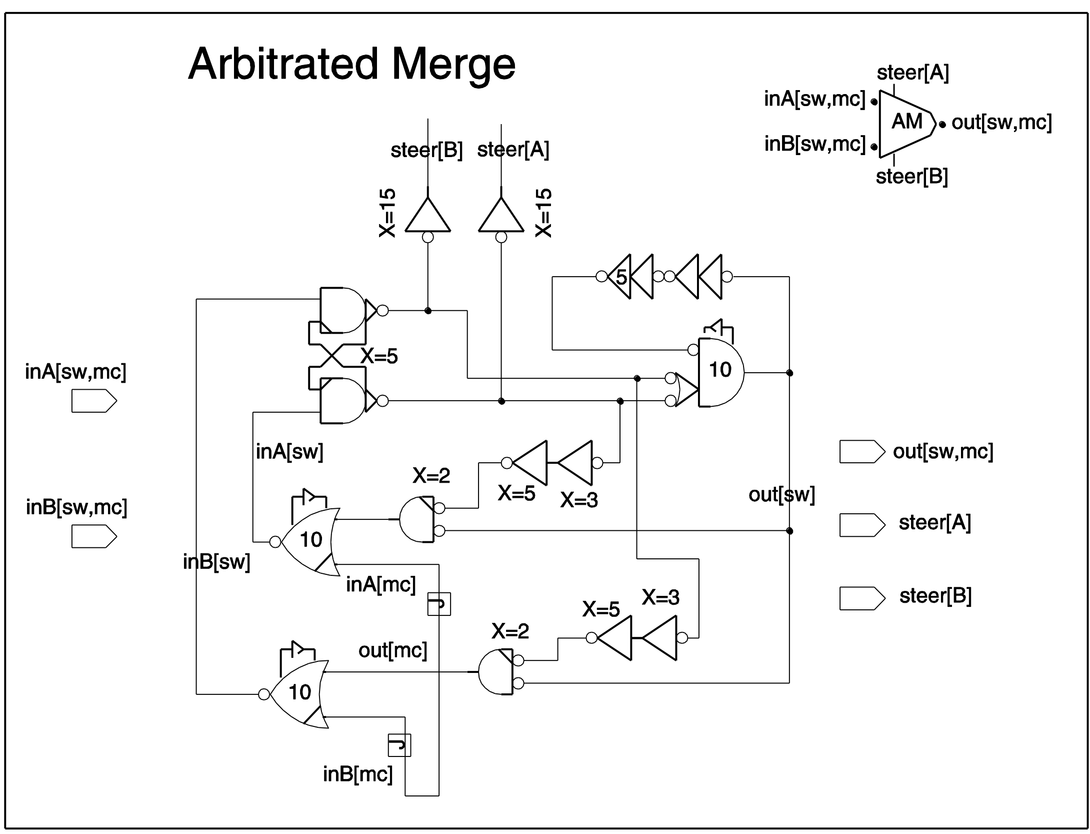


Figure 17: Telescope GasP implementation for the Arbitrated Merge module. The circuit has two input statewires with a master clear signal, **inA[sw,mc]** and **inB[sw,mc]**, two corresponding data steering signals **steer[A]** and **steer[B]**, and one output statewire with a forwarded master clear, **out[sw,mc]**. We will use the icon in the top-right corner of this picture, as a gate-level representation of an Arbitrated Merge module. This particular implementation is saved under the library name TelescopeGasP_SeparateStates, to indicate that it uses separate self-resetting loops for **inA[sw]**, **inB[sw]**, and **out[sw].**

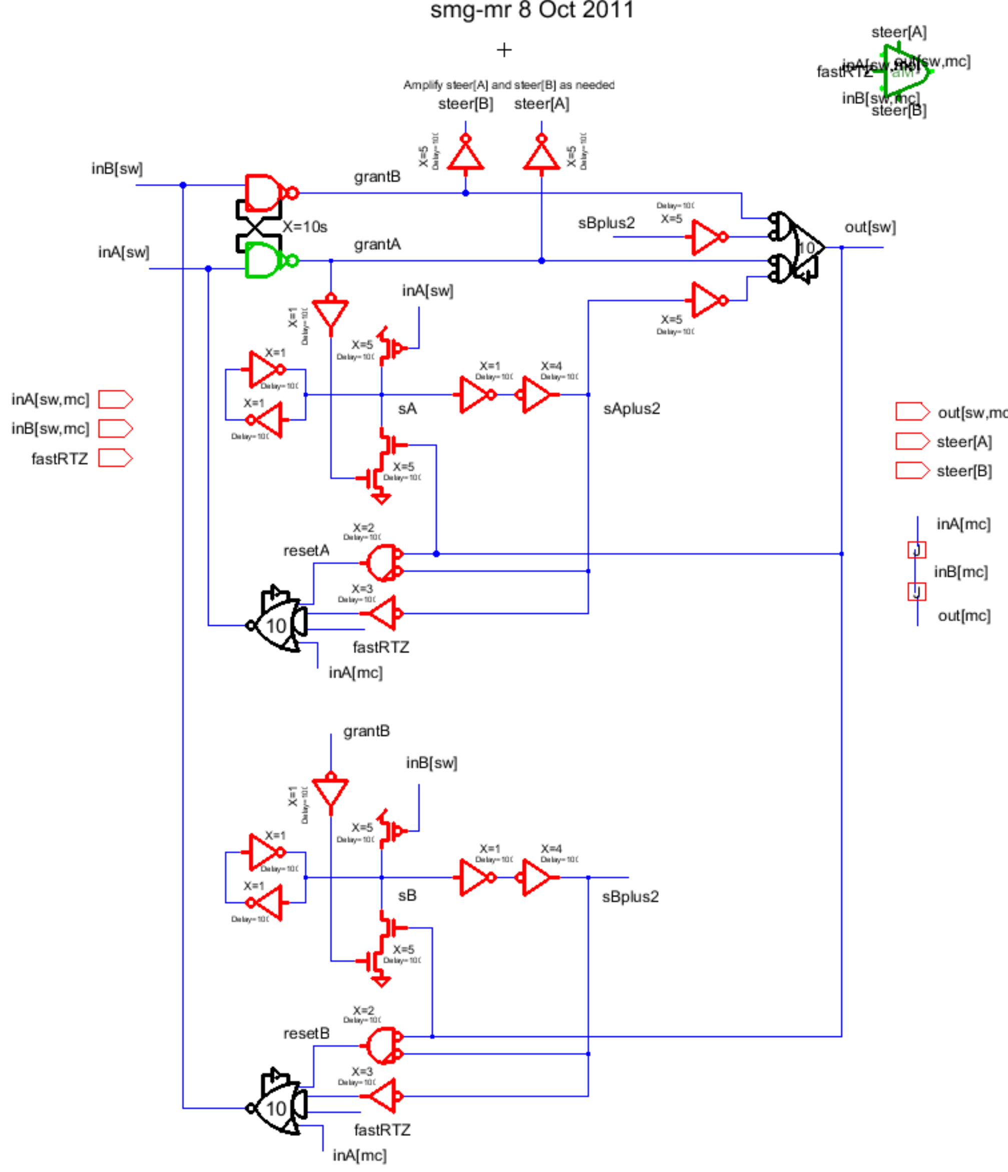


Figure 18: Second Telescope GasP implementation for the Arbitrated Merge. Here, we use a size 10 arbiter and share the self-resetting loops between incoming and outgoing statewires. In reality we need an arbiter between size 5 and 10 for both Arbitrated Merge versions. This implementation is saved in the library named TelescopeGasP_wStateSharing, under module name Merge2TfastRTZwArbiter. Substring `TfastRTZ' in the name indicates that this is a Telescope GasP implementation with fast reset capability. To ensure that we reset only the incoming handshake that caused **out[sw]** to go HI, we have AND-ed the fast reset signal at the LO driver with an internal guard signal. The guard for **inA[sw]** is tapped from internal state signal **sA** via three inversions, The guard for **inB[sw]** is tapped from internal state signal **sB** via three inversions.

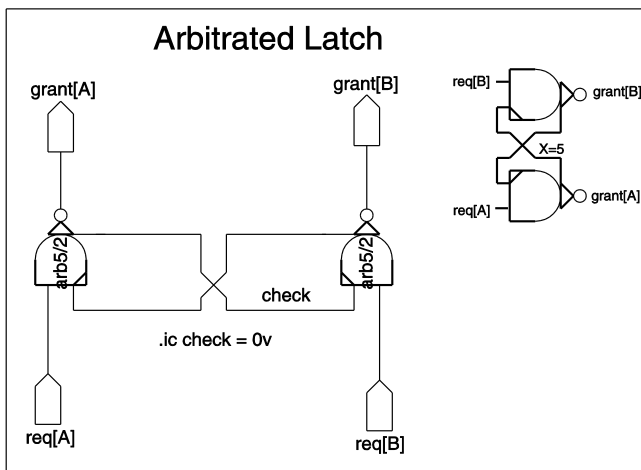


Figure 19: Circuit design of the 2-way arbiter in **Figure 17**, called Arbitrated Latch. The Arbitrated Latch is implemented as a cross-coupled pair of so-called Arbitrated Latch Nand gates, whose design follows below in **Figure 21**. When one or both of the requesting inputs **req[A]** and **req[B]** are HI, the arbiter grants exactly one of the HI requests by lowering the corresponding grant output. It lowers **grant[A]** if it grants a HI **req[A]** request, and it lowers **grant[B]** if it grants a HI **req[B]** request. It never lowers both **grant[A]** and **grant[B].** We use the icon in the top-right corner as a gate-level representation for this 2-way arbiter design.

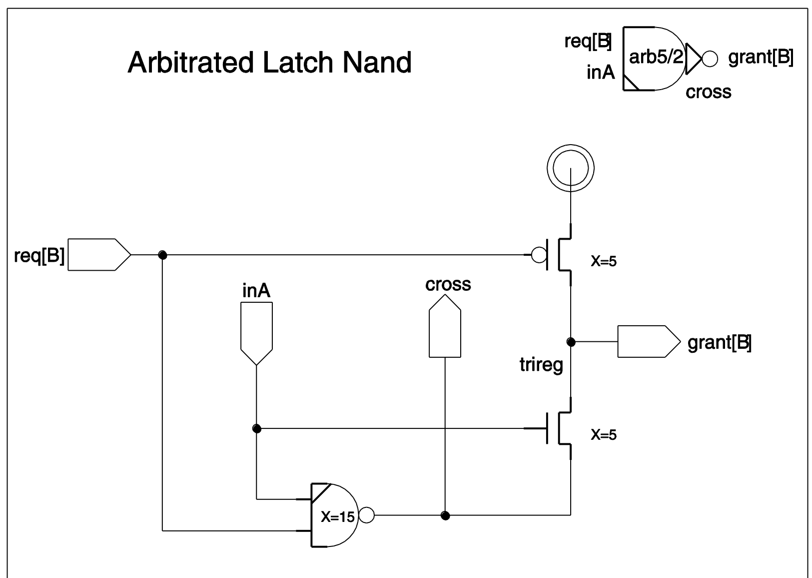
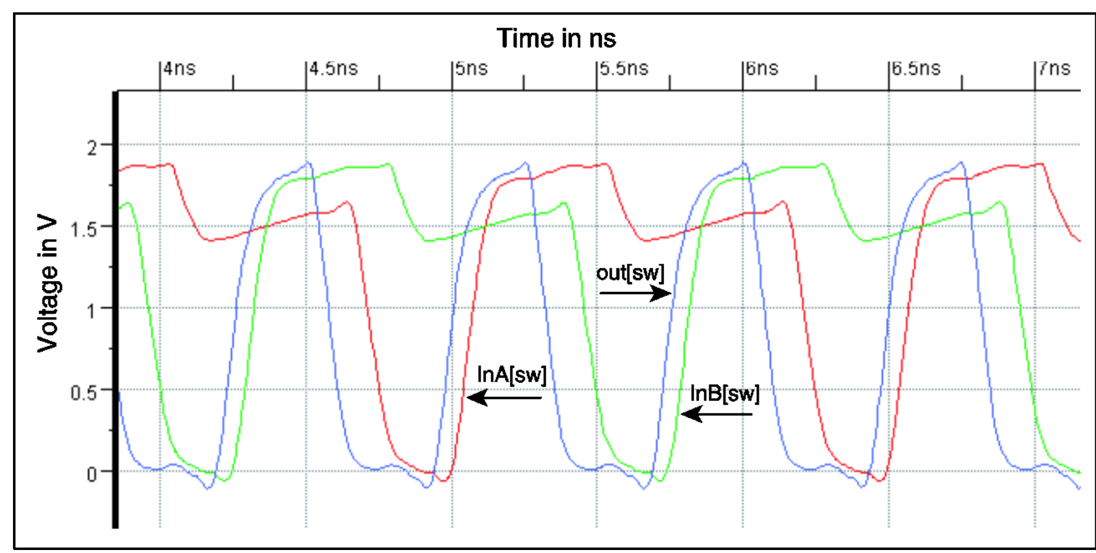


Figure 20: Circuit implementation for the Arbitrated Latch Nand circuit used in the arbiter design of **Figure 19**. Input **req[B]** is one of the input requests to the arbiter. Input **inA**, also called **cross[A]**, is cross-coupled to the intermediary output signal of the complementary Arbitrated Latch Nand gate in the arbiter. Likewise, output **cross**, also called **cross[B]**, is cross-coupled to the intermediary input signal of the complementary Arbitrated Latch Nand gate in the arbiter. Output signal **grant[B]** is driven high whenever **req[B]** is LO. If **req[B]** is HI the cross-coupling works like an arm-wrestling match between the requests. Due to the extra voltage build-up across the NMOS pass transistor between **cross** and **grant[B]**, even a mid-way voltage tie between **in[A]** and **cross** leaves both grant signals at a sufficiently high voltage level,. The combination of cross-coupling and NMOS transistor ensures that the outgoing grant signal, here **grant[B]**, retains a high voltage level until the arbitration process resolves in favor of **grant[B]**.



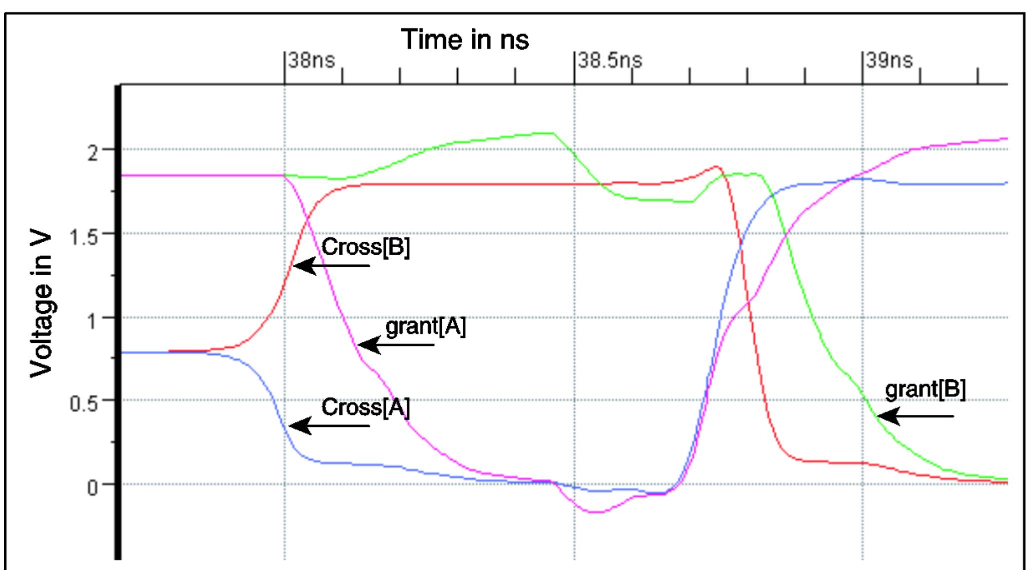


Figure 21: SPICE-level simulations of (top) the GasP implementation of the Arbitrated Merge in **Figure 17**, and (bottom) the resolution of two competing arbiter inputs and corresponding changes in voltage levels for intermediary signals and output signals of the Arbitrated Latch design in **Figure 19** and **Figure 21**.

- In the top window, one can see the fairness property of the arbiter: when statewires inA[sw] and inB[sw] arrive at about the same time and the arbiter selects inA[sw], then it will select inB[sw] next.
- The bottom window gives a more detailed view of such a scenario, including the initial metastable behavior where the arbiter is still deciding whether to select the A side or the B side.
- We are not quite happy with the voltage drop in the arbiter selected input, visible in top window. The voltage of inA[sw] and inB[sw] drops below 1.5 Volt. We can diminish this voltage drop at the cost of a somewhat slower minimum latency through the arbiter. Swetha developed a new arbiter design with a sufficiently small voltage drop for the arbiter selected input. We use it in the second Arbitrated Merge design of **Figure 18**. The resulting waveforms are visible in **Figure 15** and **Figure 16**, and show very stable and strong voltage levels. An analysis of the issues and presentation of the new arbiter design solution will be published as a separate ARC report [12].

Internally, the behaviors of the two Arbitrated Merge implementations differ. The advantage of the implementation in **Figure 18** is that it has delay margins that are relatively safe, whereas the implementation in **Figure 17** has two delay margins that are marginal. The differences in delay margin are very similar to those described in Section 4.1 for the two implementations of the Merge module.

The two marginal delay margins in **Figure 17** are as follows:

1. After either *inA[sw]* or *inB[sw]* have been selected by the arbiter, it takes 1 more gate delay before *out[sw]* rises and disables the two inverted input AND gates in the two lower self-resetting loops, while it takes 2 more gate delays for the selected signal to enable its corresponding AND gate via its lower self-resetting loop. To guarantee a telescope relation between *inA[sw]* and *out[sw]* and between *inB[sw]* and *out[sw]*, the inverted input AND gate must sense the disabling *out[sw]* transition before it senses the enabling incoming transition. The delay margin is only 1 gate delay.
2. After *out[sw]* falls, it takes 2 gate delays before the present HI signal on either *inA[sw]* or *inB[sw]* falls and disables the NAND gate in the upper self-resetting loop, while it takes 3 gate delays for the LO *out[sw]* signal to enable that same NAND gate via the upper self-resetting loop. To guarantee a correct handshake relation between *inA[sw]* and *out[sw]* and between *inB[sw]* and *out[sw],* the NAND gate must sense the disabling incoming transition before it senses the enabling *out[sw]* transition. The delay margin between the two is only1 gate delay

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Arbitrated module in **Figure 17** as safe as those in **Figure 18**.

The two implementations have the same cycle time and the same latencies. The Forward Transfer parts in **Figure 17** and **Figure 18** use respectively the Succ ORAND Driver of **Figure 4** and the Succ OR2xAND Driver of **Figure 5**. There is 1 additional gate, namely the arbiter, between **inA[sw]** respectively **inB[sw]** and the input of the driver in the Forward Transfer part, giving a minimal forward latency of 2 gate delays.

This minimum latency holds if the arbiter is uncontested, for instance when there is only one input request. The arbiter is a metastable device and so, in theory, when both input requests arrive at about the same time, the delay through the arbiter can be arbitrarily long. In practice, however, arbitrations are rarely contested, and their resolution time is generally around 1 gate delay.

In both implementations, the two Backward Transfer parts are implemented using a Pred OR Driver. The Pred OR Drive version used in **Figure 18** has an additional fast reset, and can be found in **Figure 9**. There is 1 additional gate between the outgoing channel **out[sw]** of each Arbitrated Merge implementation and the input of the two Pred OR Drivers, giving a total backward latency of 2 gate delays.

Without a fast reset signal, the minimum cycle time for each implementation is 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops in the circuit diagrams of **Figure 17** and **Figure 18**, assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each successor module between the Arbitrated Merge module and the following Store module.

Using a fast reset signal, we can reduce the backward latency and lower the cycle time to 12 gate delays per Arbitrated Merge module, with an increase in steps of 2 gate delays for each successor module between the Arbitrated Merge module and the following Store module. We designed a test environment that generates such a fast reset signal. The test setup and the generation of the fast reset signal, fastRTZ, are very similar to the test setup and fast reset generation for Telescope GasP module Merge. Test setup and simulation waveforms for the Arbitrated Merge follow in **Figure 22**, **Figure 23**, and

Figure 24 below.

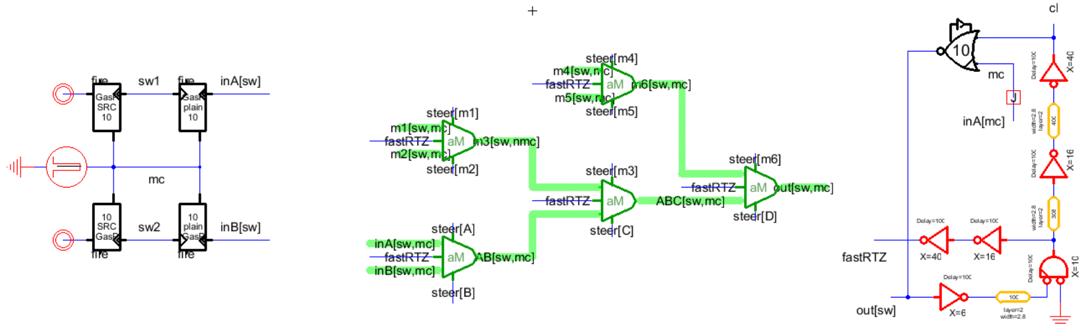


Figure 22: Fast reset configuration and test environment to concurrently reset all Arbitrated Merge connected statewires that actively participate in the current handshake telescope leading to out[sw] going HI. Note that, unlike the fast reset signal **mc_fastRTZ** in [9], but like the fast reset signal **fastRTZ** in the Default Merge module in this document, the fast reset signal **fastRTZ** generated by the Store module in this picture is independent of master clear signal **mc**.

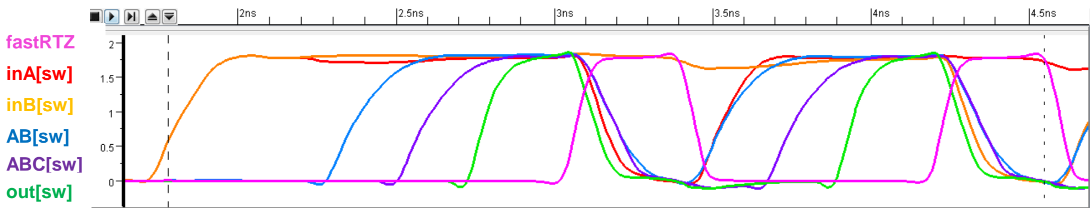


Figure 23: SPICE-level simulation results for the fast reset and test configuration in **Figure 22**. The simulation window shows how the pink HI pulse on fast reset signal fastRTZ concurrently resets all handshake signals that caused out[sw] to rise. The first HI pulse on fastRTZ has no effect on the HI inB[sw] signal, in orange, which is exactly how it should be as inA[sw] - *and not* inB[sw] - was selected by the arbiter, and so inB[sw] did not participate in the first out[sw] handshake. Likewise, the second HI pulse on fastRTZ has no effect on the HI inA[sw] signal, in red, which is also how it should be given that inA[sw] was not arbiter selected and hence did not participate in the second out[sw] handshake. The reset slope for out[sw] starts earlier and is slightly steeper than for the other handshake signals. Details follow in

Figure 24 below.

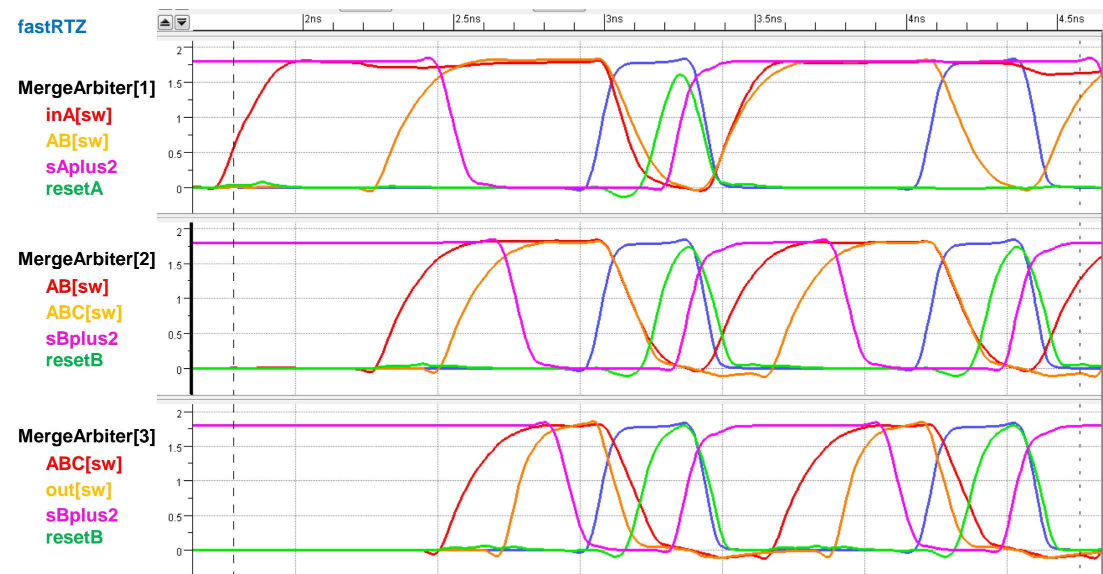


Figure 24: SPICE-level simulation details for the fast reset and test configuration in **Figure 22**. The top window contains waveforms for signals in the bottom-left Arbitrated Merge module, MergeArbiter[1]. The middle and bottom windows show the waveforms for similar signals in the bottom-middle and rightmost modules of the tree, MergeArbiter[2] and MergeArbiter[3]. The waveforms display the slope differences between the falling handshake transitions on the module's output channel, in orange, and the module's input channel, in red. The details are similar to those for the default Merge component, explained in **Figure 16**. The differences in start and steepness of the falling slopes for the statewires of each module can be explained from differences in respectively (a) input slopes to the LO drivers for the statewires and (b) stray capacitance on the statewires.

4.3. 2-Way Sequential Merge Module, or Toggle Merge

The 2-way Sequential Merge module, also called Toggle Merge or Round-Robin Merge, provides single input to single output communication. The Toggle Merge will alternate the communications over its input channels, starting with **in1[sw]**, then **in2[sw]**, then again **in1[sw]**, etcetera. For the selected incoming communication, it performs a corresponding output communication and it produces a steering signal to forward the data of the selected communication. **Figure 25** and **Figure 26** show our Telescope GasP circuit implementation for the Toggle Merge module.

The implementation in **Figure 25** is an example of our first design approach with independent self-resetting loops for statewires of incoming and outgoing handshake channels. It has four self-resetting loops to start and stop the drives for external, incoming or outgoing, statewires. The top and bottom loops are used to start and stop the HI drive for *out[sw].* The two middle loops are used to start and stop the LO drives for *inA[sw]* respectively *inB[sw].* There are two internal statewires, called *alt1[sw]* and *alt2[sw],* to keep track of whose input round it is. Their self-resetting loops overlap with the two middle loops.

The Toggle Merge implementation in **Figure 26** is an example of the second design approach with shared self-resetting loops between the statewires of incoming and outgoing handshake channels. It has two self-resetting loops for external statewires, with separate taps to start and stop the HI drive for *out[sw]* and the LO drives for *inA[sw]* and *inB[sw].* It has two additional loops to encode internal statewires, called *RR1* and *RR2*, that keep track of whose input round it is.

Note that in each implementation, we tap the data steering signals from the internal statewires. This is possible, because the internal statewires remain stable until the output communication is complete.

A SPICE-level simulation of the handshake behavior of the Toggle Merge implementation in **Figure 25** follows in **Figure 27**. The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **in1[sw]**, **in2[sw]**, **out[sw]**, and **mc** are LO, and input of choice is **in1[sw]**, as indicated by **alt1[sw]** being initialized HI and **alt2[sw]** being initialized LO. When **in1[sw]** goes HI, indicating the presence of data, and **out[sw]** is LO, indicating the availability of space, the following four sets of actions are started **in sequence**, as follows:

1. The Forward Transfer section signals the presence of new data to the successor module by driving **out[sw]** HI. This takes 2 gate delays. Meanwhile, the relevant, now HI, alt-signal steers the data that are bundled with the input of choice to the output.
2. The Forward Reset section kicks in, cutting off the HI forward drive to a 5 gate delay pulse signal
3. The Backward Transfer waits until **out[sw]** is LO, and then it starts three actions simultaneously: (1) it reports availability of space to the predecessor of choice by driving the associated input LO, (2) it drives the HI alt-signal LO, and (3) it drives the complementary alt-signal HI, thus assigning the next input of choice to the other input. Each of these three actions takes 2 gate delays. The combined delay is also 2. Data signals will now be steered from the other input to the output.
4. The Backward Reset section kicks in, cutting off the LO backward drive, the LO alt-signal drive, and the HI complementary alt-signal drive to a 5 gate delay pulse signal, each.

At the end of these four **serial** sets of actions, the Toggle Merge waits until input **in2[sw]** is HI, so it can start its next cycle of parallel actions. These cycles repeat with the input of choice alternating or toggling between **in1[sw]** and **in2[sw]**.

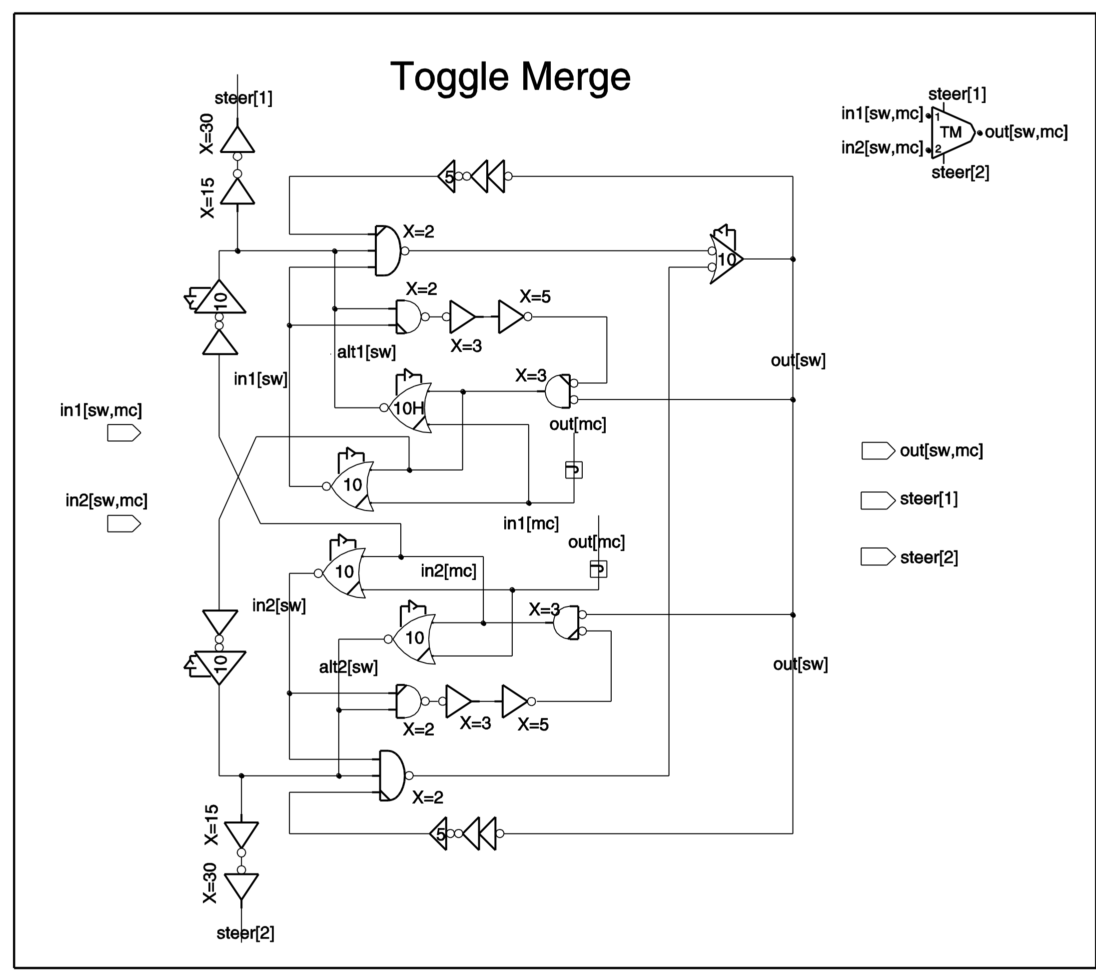


Figure 25: Telescope GasP control circuit for the 2-way Sequential Merge module, also called Toggle Merge. The circuit has two input statewires with a master clear signal, **in1[sw,mc]** and **in2[sw,mc]**, their corresponding data steering signals **steer[1]** and **steer[2]**, and one output statewire with a joined master clear, **out[sw,mc]**. The two internal statewires alt1[sw] and alt2[sw] keep track of which input is input of choice. The first round is for **in1[sw],** the next for **in2[sw],** and so on, in a round-robin fashion. The Pred Driver icon marked "10H", initializes **alt1[sw]** to HI. The circuit is thus initialized in favor of input channel **in1[sw]**. This implementation is saved under TelescopeGasP_SeparateStates to indicate that it uses separate self-resetting loops for **in1[sw]**, **in2[sw]**, and **out[sw].** We will use the icon in the top-right corner of this picture, when we represent a Toggle Merge. We will also use rrM instead of TM as icon text, where 'rr' stands for 'round-robin'.

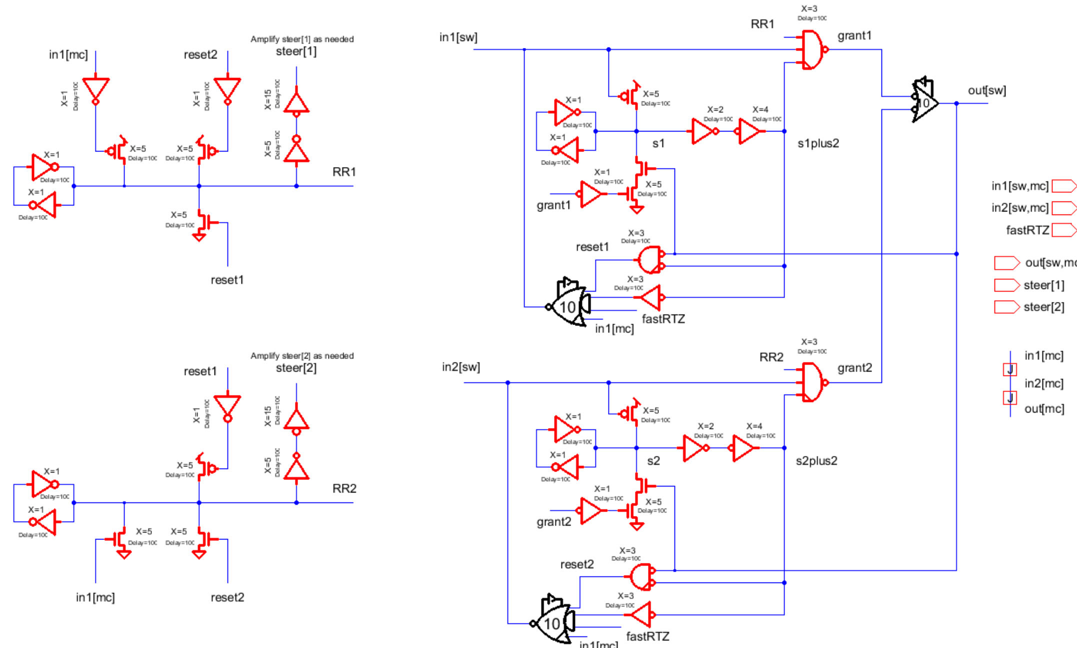


Figure 26: Second Telescope GasP implementation for the Toggle Merge. Here, we share the self-resetting loops between incoming and outgoing statewires insofar possible. This implementation is saved in the library named TelescopeGasP_wStateSharing, under module name Merge2TfastRTZwArbiter. Substring `TfastRTZ' in the name indicates that this is a Telescope GasP implementation with fast reset capability. To ensure that we reset only the incoming handshake that caused **out[sw]** to go HI, we have AND-ed the fast reset signal at the LO driver with an internal guard signal. The guard for **in1[sw]** is tapped from internal state signal **s1** via three inversions, The guard for **in2[sw]** is tapped from internal state signal **s2** via three inversions.

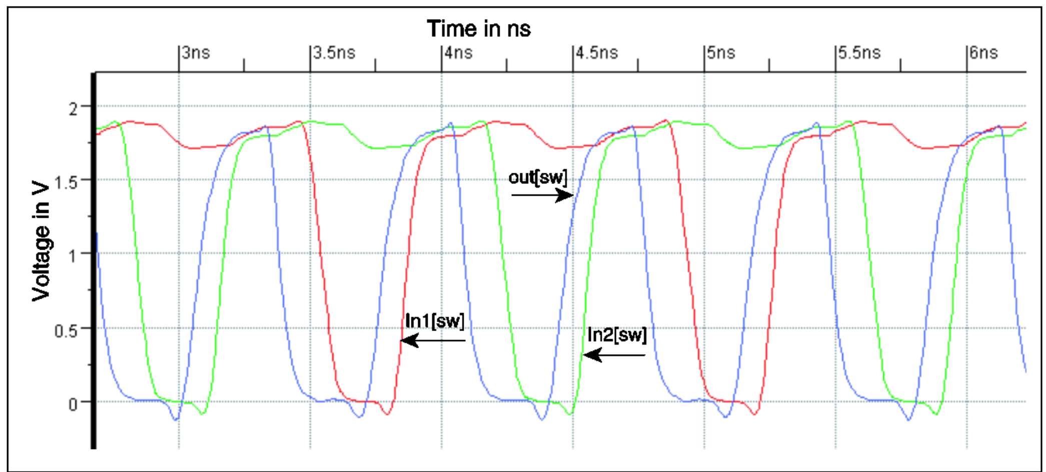


Figure 27: SPICE-level simulations of the GasP implementation of the Toggle Merge in **Figure 25**. Red and green statewires in1[sw] and in2[sw] have a telescopic handshake relationship to blue statewire out[sw]: in1[sw] or in2[sw] HI causes out[sw] to go HI, and when out[sw] has gone LO, only then will the input go LO. The simulation clearly shows that (1) each HI out[sw] pulse is strictly contained within a HI pulse on in1[sw] or in2[sw] and (2) out[sw] handshakes go alternately with in1[sw] respectively in2[sw] handshakes.

Internally, the behaviors of the two Toggle Merge implementations differ. The implementation in **Figure 26** has relatively safe delay margins, whereas the implementation in **Figure 25** has two delay margins that are marginal. The two marginal delay margins in **Figure 25** are as follows:

1. After the chosen *in1[sw]* or *in2[sw]* has risen, it takes 2 gate delays before *out[sw]* rises and disables the two inverted input AND gates in the two middle self-resetting loops, while it takes 3 gate delays for the chosen incoming signal to enable the corresponding AND gate via its middle self-resetting loop. To guarantee a telescope relation between *in1[sw]* and *out[sw]* and between *in2[sw]* and *out[sw]*, the AND gate must sense the disabling *out[sw]* transition before it senses the enabling incoming transition. The delay margin between the two is only 1 gate delay.
2. After *out[sw]* falls, it takes 2 gate delays before the present HI signal on the chosen *in1[sw]* or *in2[sw]* falls and disables the three input NAND gate in the upper or lower loop, while it takes 3 gate delays for the LO *out[sw]* signal to enable that same NAND gate via the upper or lower self-resetting loop. To guarantee a correct handshake relation between *in1[sw]* and *out[sw]* and between *in2[sw]* and *out[sw],* the NAND gate must sense the disabling incoming transition before it senses the enabling *out[sw]* transition. The delay margin between the two is only1 gate delay.

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Toggle Merge module in **Figure 25** as safe as those in **Figure 26**.

The two implementations have the same cycle time and the same latencies. The Forward Transfers in **Figure 25** and **Figure 26** use the Succ OR Driver of **Figure 2**. There is 1 extra gate between **in1[sw]** respectively **in2[sw]** and the input of the driver in the Forward Transfer part, giving a minimal forward latency of 2 gate delays. The Backward Transfer parts are implemented using a Pred OR Driver. The Pred OR Drive version used in **Figure 26** has an additional fast reset, and can be found in **Figure 9**. There is 1 additional gate between the outgoing channel **out[sw]** of each Toggle Merge implementation and the input of the two Pred OR Drivers, giving a total backward latency of 2 gate delays.

In addition to the Forward and Backward Transfer and Forward and Backward Reset sections, there is an additional alternating loop section for passing a round-robin token around to create alternating **in1[sw]** and **in2[sw]** communications. This token ring is implemented using internal communication channels. This is a control-oriented solution. An alternative implementation would be to encode the token ring in an internal data structure, using flipflops as is done in the Click backend to ARCwelder [6,13].

The control-oriented token ring solution presented in **Figure 25** has two internal single-track channels, or statewires, called **alt1[sw]** and **alt2[sw]**. Statewire **alt2[sw]** is initialized LO via a separate Pred Driver similar to the Pred Driver in [4]. Statewire **alt1[sw]** is initialized HI via the special Pred Driver in **Figure 25** with icon text “10H” (suffix `H’ stands for `HI’). This special Pred Driver is like the Pred Driver in [4], except that a HI master clear signal **mc** will initialize the Pred Driver output to HI. In reality, this HI master clear feature would likely be implemented as part of the HI driver and LO half-keeper logic for **alt1[sw]**.

Without a fast reset signal, both implementations have a minimum cycle time of 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops - assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each Telescope GasP successor module between the Toggle Merge module and the following Store module.

With a fast reset signal, we can reduce the backward latency and lower the cycle time to 12 gate delays per Toggle Merge module, with an increase in steps of 2 gate delays for each successor module between the Toggle Merge module and the following Store module. We designed a test environment that generates such a fast reset signal. The test setup and the generation of the fast reset signal, fastRTZ, are very similar to the test setup and fast reset generation for the previous Merge modules. Test setup and simulation waveforms for the Toggle Merge follow in **Figure 28** and **Figure 29** below.

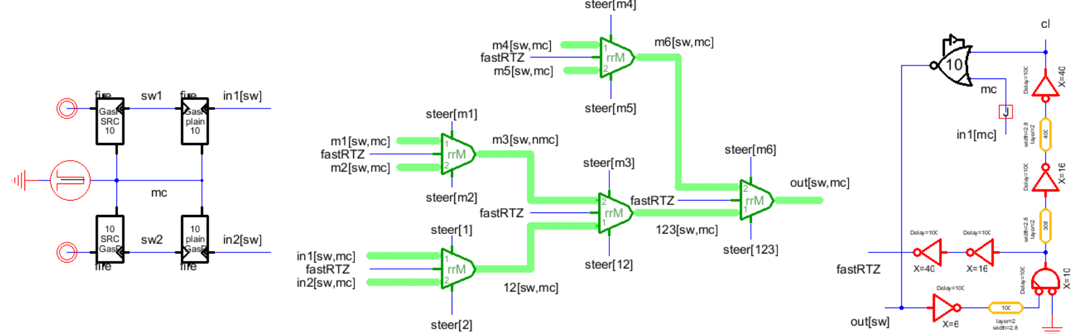


Figure 28: Fast reset configuration and test environment to concurrently reset all Toggle Merge statewires that actively participate in the current handshake telescope leading to **out[sw]** going HI. Unlike the fast reset signal **mc_fastRTZ** in [9], but like the fast reset signal **fastRTZ** in the other Merge modules in this document, the fast reset signal **fastRTZ** generated by the Store module in this picture is independent of master clear signal **mc**.

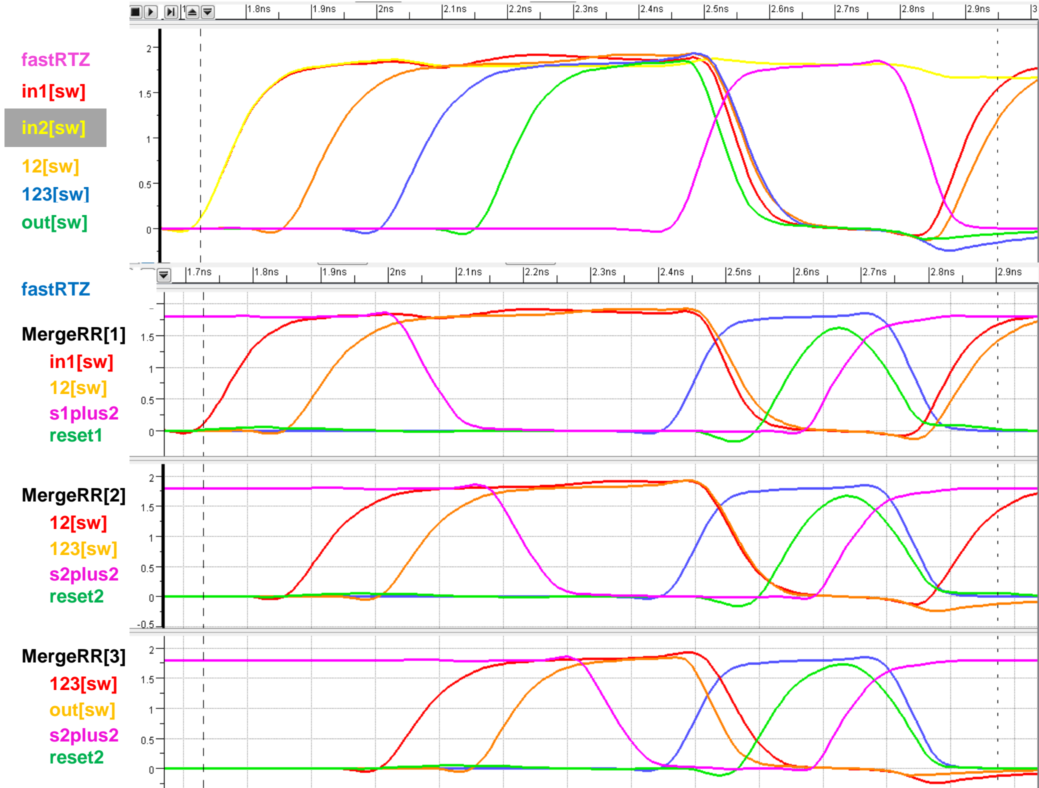


Figure 29: SPICE-level simulation results for the fast reset and test configuration in **Figure 28**. The top simulation window shows how the pink HI pulse on fast reset signal **fastRTZ** concurrently resets all handshake signals that caused the green out[sw] signal to rise. The first HI pulse on fastRTZ has no effect on the HI in2[sw] signal, in yellow, which is exactly how it should be as in1[sw] - *and not in2[sw]* – is the first input of choice. The bottom three simulation windows give a module-by-module view of the fast reset cycle. Just like for the other Merge modules, the differences in start and steepness of the falling slopes for the statewires of each Toggle Merge module can be explained from differences in (a) input slopes to the LO drivers for the statewires and (b) stray capacitance on the statewires.

4.4. 3-Way Sequential Merge Module

The 3-way Sequential Merge module or 3-way Round-Robin Merge provides single input to single output communication. It sequentially selects communications over its input channels, starting with **in1[sw]**, then **in2[sw]**, then **in3[sw]**, and back in a round-robin fashion. For the selected incoming communication, it performs a corresponding output communication on **out[sw]** and produces a steering signal to forward the data of the selected communication to **out[sw]**. **Figure 30** shows our Telescope GasP implementation for the three-way sequential Merge module.

The implementation in **Figure 30** is an example of our first design approach, with independent self-resetting loops for statewires of incoming and outgoing handshake channels. It generalizes the implementation in **Figure 25**.

The generalization from a two-way to a three-way implementation has a price. For one, there are now six instead of four self-resetting loops to start and stop the drives for statewires of external, incoming or outgoing handshake channels. Second, there are now three instead of two extra internal statewires with self-resetting loops that partially overlap the self-resetting loops for the external signals. Third, instead of the Suc OR Driver of **Figure 2** the 3-way version uses the Suc OR3 Driver of **Figure 3** to drive **out[sw].**

A similar generalization would also work for the second design approach, at the cost of a Suc OR3 Driver and two additional internal statewires: one for in3[sw] to out[sw] telescope handshakes and the other to encode a third round-robin signal, RR3.

We refrain from providing a behavioral description and waveform. The behavior is a straightforward extension of the behavior for the 2-way sequential Merge. The main motivation for designing and presenting this 3-way sequential Merge is to get and give a feel for the growth in design complexity when we generalize the basic 1of2-to-1 solution to 1ofN-to-1 solutions, for N larger than 2.

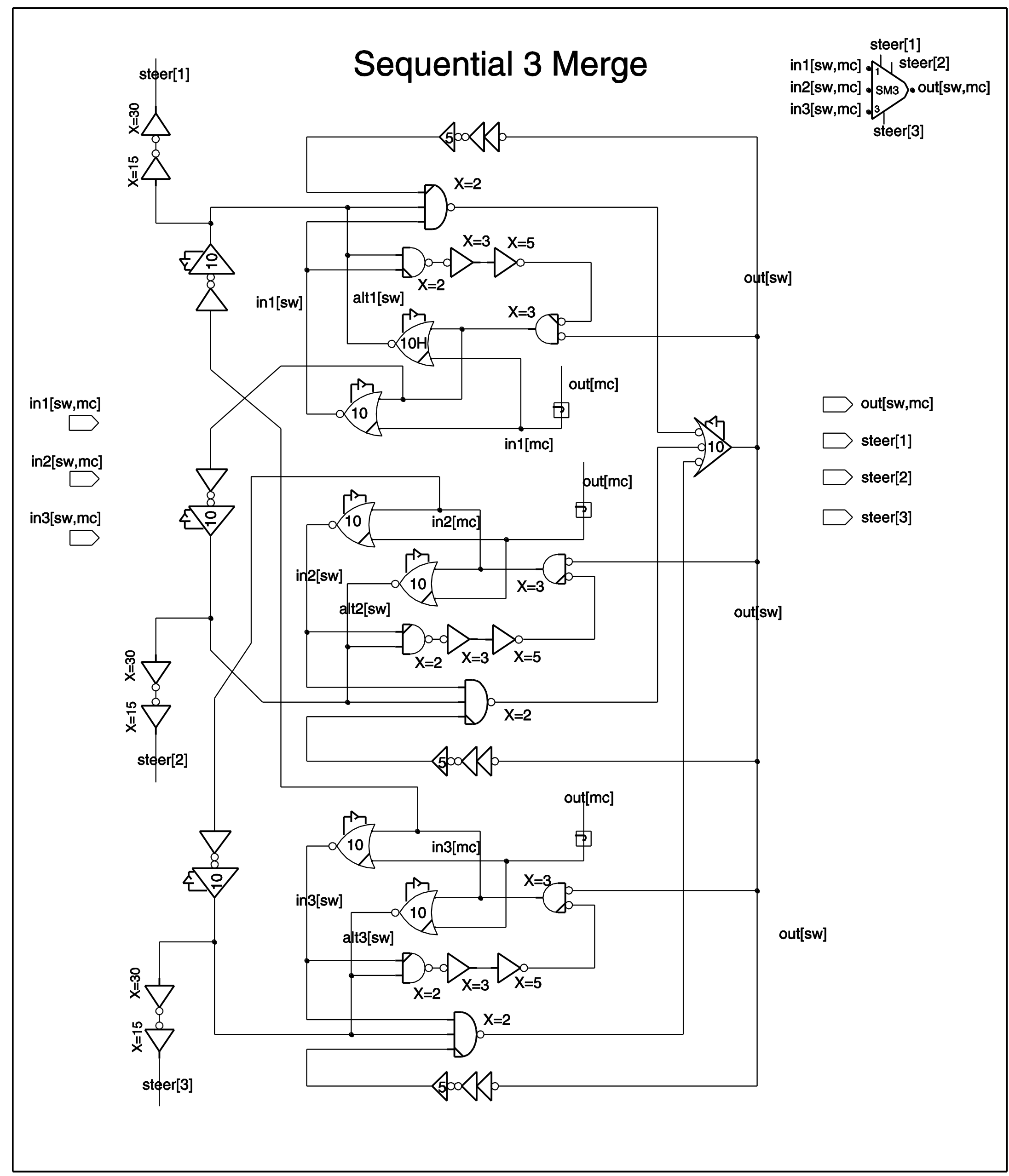


Figure 30: Telescope GasP control circuit for the 3-way Sequential Merge or Round-Robin Merge module. The circuit has three input statewires with a master clear signal, **in1[sw,mc] in2[sw,mc]**, and **in3[sw,mc]**, **]**, their corresponding data steering signals **steer[1]**, **steer[2]**, **steer[3]**, and one output statewire with a joined master clear, **out[sw,mc]**. The three internal statewires, **alt1[sw]**, **alt2[sw]**, and **alt3[sw]**, keep track of whose input turn or round it is. The first round is for **in1[sw],** the next for **in2[sw],** the one after that for **in3[sw],** and so on, in a round-robin fashion. The implementation is a straightforward extension of the 2-way Sequential merge presented in **Figure 25**. We will use the top-right icon to represent a 3-way Sequential Merge.

5. Branch Modules

Branch modules provide single input to single output 1-to-1ofN communication, by selecting exactly one out of one or more available outputs. For now, our selections are done in a sequential, or round-robin, fashion, with the only freedom being the number of module outputs. The module waits until the input communication becomes available and streams it to the corresponding output communication. We opted to broadcast the input data to all outputs, no matter whether or not the receiving module participates in the action. As a result, no special steering logic is needed. Section 5.1 below gives a GasP implementation for the 2-way Sequential Branch module with two module outputs, also known as Toggle Branch. Section 5.2 extends this GasP implementation to create a 3-way Sequential Branch module, and Section 5.3 carries this one step further to a 4-way Sequential Branch.

5.1. 2-Way Sequential Branch Module, or Toggle Branch

The 2-way Sequential Branch module, also called Toggle Branch, provides single input to single output round-robin 1-to-1of2 communication. Upon receiving an input communication, it streams the input data to one of the outputs, starting with **out1[sw]**. The next time it receives an input communication, it will pass the communication on to **out2[sw]**, then back again to **out1[sw]**, etcetera, in a round-robin fashion. **Figure 31** and **Figure 32** show our two Telescope GasP implementations for the Toggle Branch module. The implementations are more or less the reverse of the Toggle Merge in **Figure 25** and **Figure 26**.

The implementation in **Figure 31** is an example of our first design approach with independent self-resetting loops for statewires of incoming and outgoing handshake channels. It has four self-resetting loops to start and stop the drives for external, incoming or outgoing, statewires. The bottom and top loops are used to start and stop the HI drive for *out1[sw]* respectively *out2[sw],* and the two middle loops are used to start and stop the LO drive for *in[sw].* The other two loops encode internal statewires, called *alt1[sw]* and *alt2[sw],* to keep track of whose output round it is – these partially overlap with the two middle loops for *in[sw].*

The Toggle Branch implementation in **Figure 32** is an example of the second design approach with shared self-resetting loops between the statewires of incoming and outgoing handshake channels. It has two self-resetting loops for external statewires, with separate taps to start and stop the HI drives for *out1[sw]* and *out2[sw]* and the LO drive for *in [sw].* It has two additional loops to encode internal statewires, called *RR1* and *RR2*, to keep track of whose output round it is.

SPICE-simulated waveforms of the handshake behavior of the implementation in **Figure 31** follow in **Figure 33**. The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **in[sw]**, **out1[sw]**, **out2[sw]**, and **mc** are LO, **alt1[sw]** is HI, and **alt2[sw]** is LO. When **in[sw]** goes HI, indicating it has data, and output **out1[sw]** is LO, indicating it has space to receive these data, and **alt1[sw]** is HI, indicating that **out1[sw]** is the present output of choice, the following four actions are started **sequentially**, in order of occurrence:

1. The Forward Transfer part signals the presence of new data to the successor module of choice by driving the output of choice HI. This takes 2 gate delays.
2. The Forward Reset kicks in, cutting off the HI forward drive to a 5 gate delay pulse signal.
3. The Backward Transfer waits until the chosen output is LO, and then it starts three actions in parallel: (1) it reports availability of space to the predecessor by driving **in[sw]** LO, (2) it drives the HI alt-signal LO, and (3) it drives the complementary alt-signal HI, thus assigning the next output of choice to the other output. Each of these three actions takes 2 gate delays. Given that these three actions take place in parallel, the total execution time is also 2 gate delays.
4. The Backward Reset section kicks in, cutting off the LO backward drive, the LO alt-signal drive, and the HI complementary alt-signal drive to a 5 gate delay pulse signal, each.

At the end of these four **serial** actions, the Toggle Branch waits until input **in[sw]** is HI again, so it can start its next cycle of parallel actions. These cycles repeat with the output of choice alternating between **out1[sw]** and **out2[sw]**.

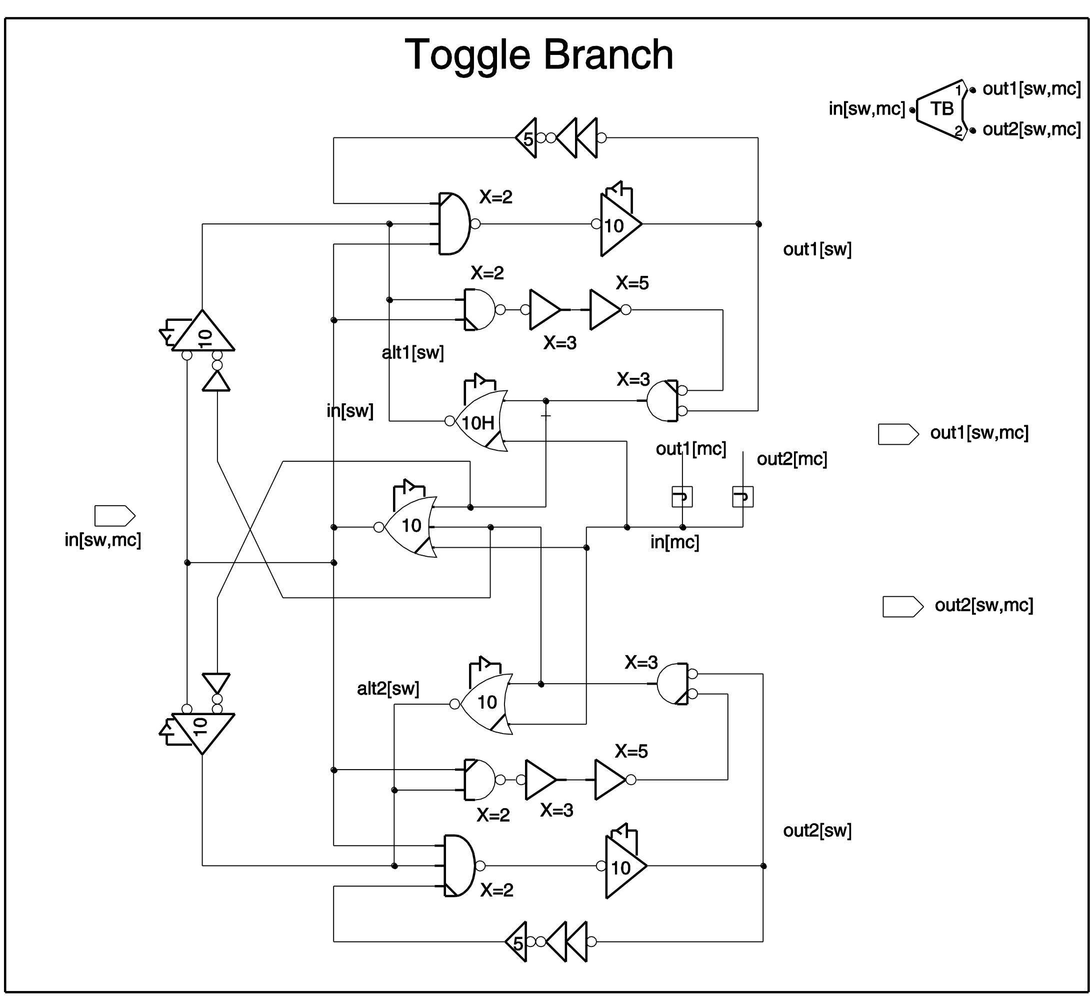


Figure 31: Telescope GasP implementation for the 2-way Sequential Branch module, also called Toggle Branch. This implementation is more-or-less the reverse of the Toggle Merge circuit in **Figure 25**. The circuit has one input statewire with a master clear signal, **in[sw,mc]**, and two output statewires with a forwarded master clear signal, **out1[sw,mc]** and **out2[sw,mc]**. The two internal statewires **alt1[sw]** and **alt2[sw]** serve to keep track of whose output turn or round it is. The first round is for **out1[sw],** the next for **out2[sw],** and so on, in a round-robin fashion. The Pred Driver icon marked "10H", initializes **alt1[sw]** to HI. The circuit is thus initialized in favor of output channel **out1[sw]**. This particular implementation is saved under the library TelescopeGasP_SeparateStates to indicate that it uses separate self-resetting loops for **in1[sw]**, **in2[sw]**, and **out[sw].** We will use the icon in the top-right corner of this picture, when we represent a Toggle Branch module. We may also use rrB instead of TB as icon text, where 'rr' stands for 'round-robin'.

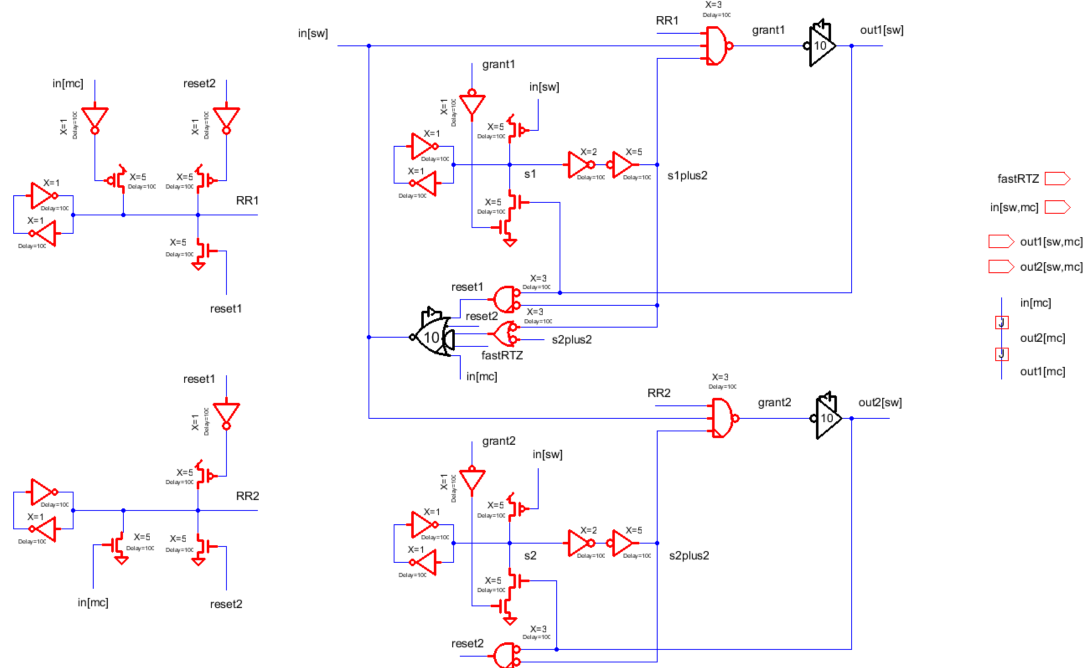


Figure 32: Second Telescope GasP implementation for the Toggle Branch, also called Round-Robin Branch. Here, we share the self-resetting loops between incoming and outgoing statewires insofar possible. This implementation is saved in the library named TelescopeGasP_wStateSharing, under module name BranchRR2TfastRTZ. Substring RR2 in the name indicates that this is a round-robin version with 2 output. Substring `TfastRTZ' indicates that this is a Telescope GasP implementation with fast reset capability. Internal statewires RR1 and RR2 keep track of whose output round it is. Internal statewires s1 and s2 control the begin and end of the HI drive for outgoing statewires and the LO drive for the incoming statewire.

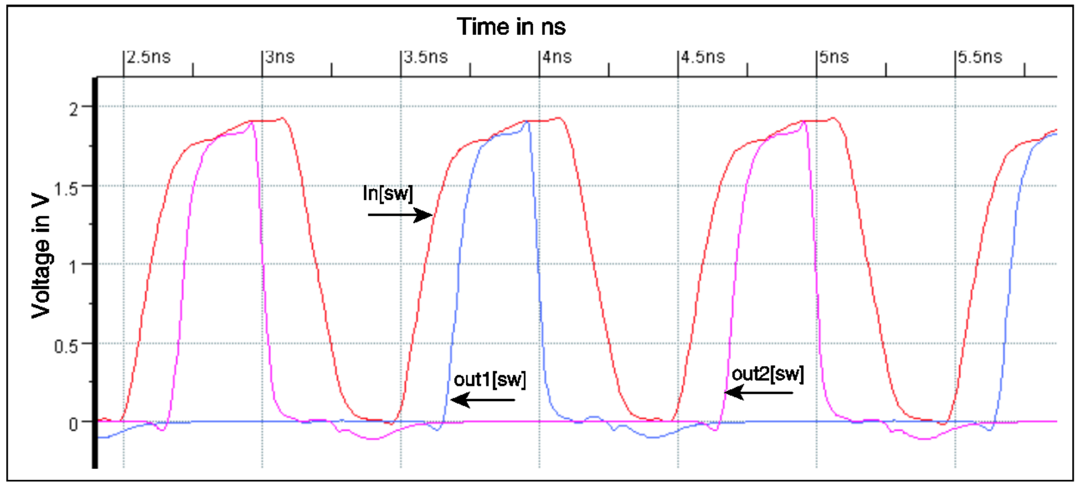


Figure 33: SPICE-level simulations of the GasP implementation of the 2-way Sequential Branch in **Figure 31**. Red statewire in1[sw] has a telescopic handshake relationship to blue and pink statewires ou1t[sw] and out2[sw]: in[sw] HI causes either out1[sw] or out2[sw] to go HI. When the respective output has gone LO, only then will in[sw] go LO. The simulation clearly shows that (1) each HI output pulse is strictly contained within a HI pulse on in[sw], and (2) in[sw] handshakes telescope alternately with out1[sw] and out2[sw] handshakes.

Internally, the behaviors of the two implementations differ. The implementation in **Figure 32** has relatively safe delay margins, whereas the implementation in **Figure 31** has two delay margins that are marginal. The two marginal delay margins in **Figure 31** are as follows:

1. After *in[sw]* has risen, it takes 2 gate delays before the chosen *out1[sw]* or *out2[sw]* rises and disables the two inverted input AND gate connections to the two middle self-resetting loop, while it takes 3 gate delays for the chosen incoming signal to enable the corresponding AND gate via its middle self-resetting loop. To guarantee a telescope relation between *in1[sw]* and *out[sw]* and between *in2[sw]* and *out[sw]*, the AND gate must sense the disabling *out[sw]* transition before it senses the enabling incoming transition. The delay margin between the two is only 1 gate delay.
2. After the chosen *out1[sw]* or *out2[sw*] falls, it takes 2 gate delays before *in[sw]* falls and disables the two three input NAND gate in the top and bottom loops, while it takes 3 gate delays for the chosen outgoing signal to enable that same NAND gate via its top or bottom self-resetting loop. For a correct handshake relation between *in[sw]* and *out1[sw]* respectively *in[sw]* and *out2[sw],* the NAND gate must sense the disabling transition before it senses the enabling one. The delay margin between the two is only1 gate delay.

A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the Toggle Branch module in **Figure 31** as safe as those in **Figure 32**.

The two implementations have the same cycle time and the same latencies. The Forward Transfers in **Figure 31** and **Figure 32** use the Succ Driver in [9]. There is 1 extra gate between **in[sw]** and the input of the Succ Driver, giving a minimal forward latency of 2 gate delays. The Backward Transfer parts are implemented using a Pred OR Driver. The Pred OR Driver version used in **Figure 32** has an additional fast reset, and can be found in **Figure 9**. There is 1 additional gate between outgoing channel **out1[sw]** and **out2[sw]** of each Toggle Branch implementation and the input of the corresponding Pred OR Driver, giving a total backward latency of 2 gate delays.

In addition to the Forward and Backward Transfer and Forward and Backward Reset parts, there is an additional alternating loop section for passing a round-robin token around to create alternating **out1[sw]** and **out2[sw]** communications. This token ring is implemented using internal communication channels. Just like in the earlier Toggle Merge implementations, these are control-oriented solutions. Alternative implementations can be obtained by encoding the token ring in an internal data structure, for instance by using flipflops as is done in the Click backend to ARCwelder [6,13].

The control-oriented token ring solution presented in **Figure 31** has two internal single-track channels, or statewires, called **alt1[sw]** and **alt2[sw]**. Statewire **alt2[sw]** is initialized LO via a separate Pred Driver similar to the Pred Driver in [9]. Statewire **alt1[sw]** is initialized HI via the special Pred Driver in **Figure 31** with icon text “10H” (suffix `H’ stands for `HI’). This special Pred Driver is like the Pred Driver in [9], except that a HI master clear signal **mc** will initialize the Pred Driver output to HI. This HI master clear feature could be - and preferably will in the future - part of the HI driver and LO half-keeper logic for **alt1[sw]**.

Without a fast reset signal, both implementations have a minimum cycle time of 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops - assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each Telescope GasP successor module between the Toggle Branch module and the following Store module.

With a fast reset signal, we can reduce the backward latency and lower the cycle time to 12 gate delays per Toggle Branch module, with an increase in steps of 2 gate delays for each successor module between the Toggle Branch module and the following Store module. We designed a test environment that generates such a fast reset signal. The test setup and the generation of the fast reset signal, fastRTZ, are similar to the test setup and fast reset generation for the Merge modules. Test setup and simulation waveforms for the Toggle Branch follow in **Figure 34** and **Figure 35** below.

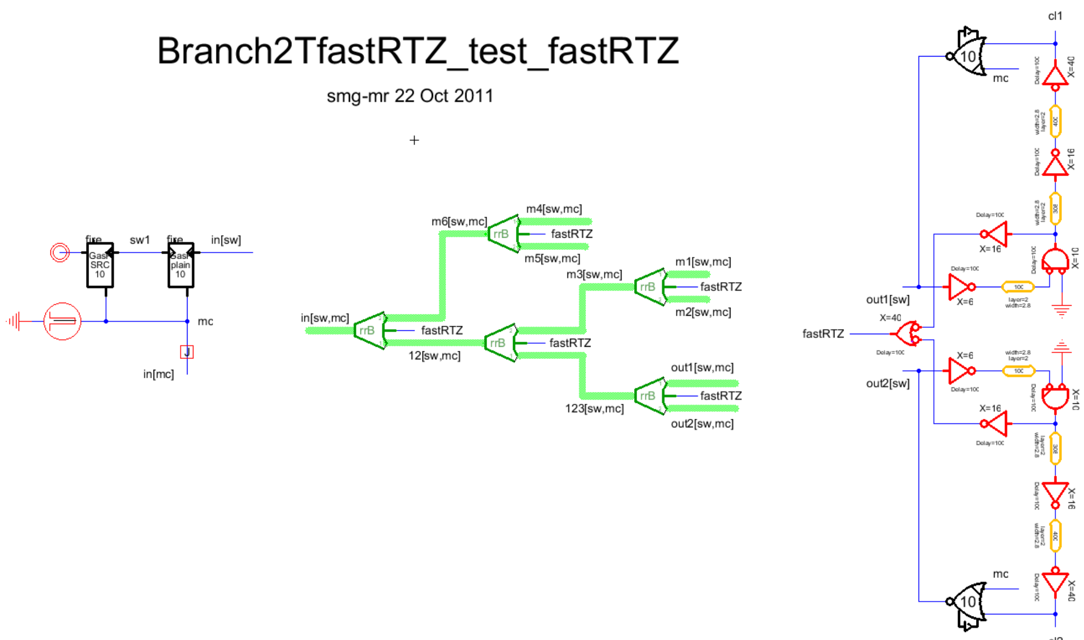


Figure 34: Fast reset configuration and test environment to concurrently reset all Toggle Branch statewires that actively participate in the current handshake telescope leading to **out1[sw]** or **out2[sw]** going HI. Unlike fast reset signal **mc_fastRTZ** in [9], but like the fast reset **fastRTZ** used in the Merge modules in this document, the fast reset signal **fastRTZ** generated by the Store module in this picture is independent of master clear signal **mc**. Though hard to see, some of the Toggle Branch icons in this picture are flipped over their horizontal axis. As a result, the first telescope handshake takes the path (in[sw],12[sw],123[sw],out1[sw]).

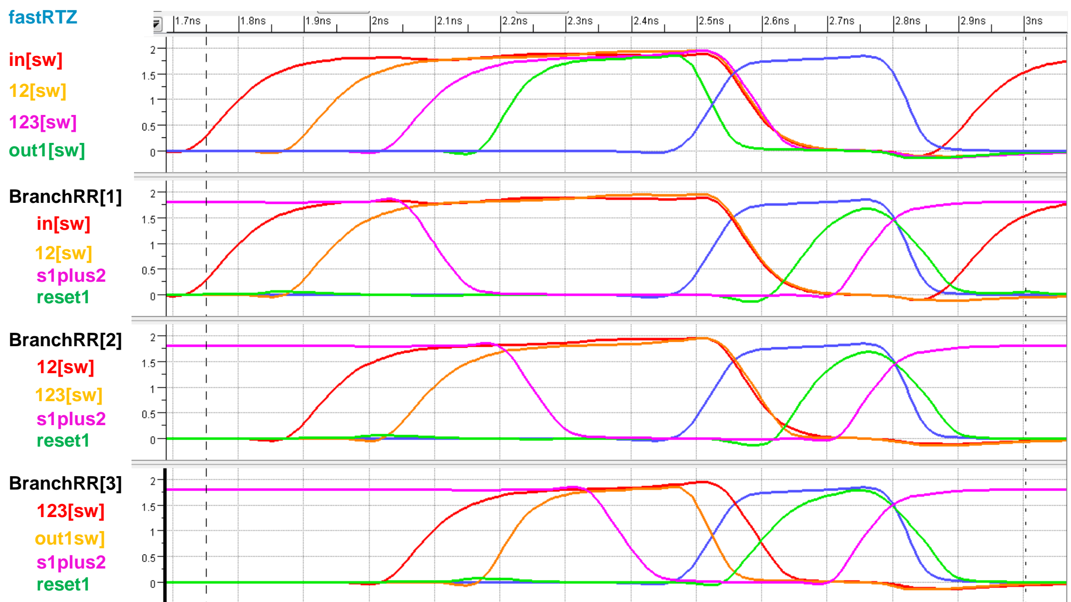


Figure 35: SPICE simulation results for the first telescope handshake over (in[sw],12[sw],123[sw],out1[sw]) in the fast reset and test configuration in **Figure 34**. The top window shows how the blue HI pulse on fast reset signal fastRTZ concurrently resets all handshake signals that caused out1[sw] to rise. During the 5 gate delay HI pulse on fastRTZ, all orange and red handshake signals are reset to LO, simultaneously. The bottom three simulation windows show a module-by-module snapshot of the fast reset cycle. Just like was the case for the Merge modules, the differences in start and steepness of the falling slopes for the statewires of each Toggle Branch module can be explained from differences in (a) input slopes to the LO drivers for the statewires and (b) stray capacitance on the statewires.

5.2. 3-Way Sequential Branch Module

The 3-way Sequential Branch module provides single input to single output communication. Whenever it receives a communication over its input channel, **in[sw]**, it streams the corresponding data to one of the output channels, **out1[sw]**, **out2[sw]**, or **out3[sw]**. It selects the output communication channel in a round-robin fashion, starting with **out1[sw]**.

Figure 36 gives our Telescope GasP circuit implementation for the 3-way Sequential Branch. The circuit implementation is a straightforward extension of that for the 2-way Sequential Branch, or Toggle Branch in **Figure 31**. The token ring now contains three statewires: **alt1[sw]**, **alt2[sw]**, and **alt3[sw]**. The three statewires are connected so that the single HI "token" goes around in a round-robin fashion.

The generalization from a two-way to a three-way implementation has a price. For one, there are now six instead of four self-resetting loops to start and stop the drives for statewires of external, incoming or outgoing handshake channels. Second, there are now three instead of two extra internal statewires with self-resetting loops that partially overlap the self-resetting loops for the external signals. Third, instead of the Pred OR Driver of **Figure 6** the 3-way version uses the Pred OR3 Driver of **Figure 7** to drive **in[sw].**

A similar generalization would also work for the second design approach, at the cost of two additional internal statewires: one for **in[sw]** to **out3[sw]** telescope handshakes and the other to encode a third round-robin signal, RR3. We would also need a Pred OR3 Driver with both a master clear and a fast reset, This predecessor driver is where the bottleneck lies: the implementation would require a stack of six levels of PMOS transistors in series to implement the HI keeper part of this drive-LO and keep-HI device. We are hesitant to use such deep transistor stacks.

There are other ways to build N-way Sequential Branch modules:

- Instead of adding functional complexity to a single module, we could distribute the complexity over multiple modules, like we do for the four-way Sequential Branch in the subsequent Section 5.3.
- We could relax the 2 gate delay forward and backward latencies in more complex N-way module designs, where N exceeds 2. This would enable us to spread the complexity over multiple gates and drivers, and keep the gates and drivers relatively simple. A module with a forward and backward latency of 3 or 4 gate delays would still have a smaller or equal latency compared to a design with two or more modules in series.

Notes:

- The first design approach would also need a Pred OR3 Driver if it were provisioned with a fast reset.
- We do not provide a behavioral description and waveform for the 3-way Sequential Branch. The behavior is a straightforward extension of the behavior for the 2-way sequential Branch. The main motivation for designing and presenting this 3-way sequential Branch is to get a feel for the growth in design complexity when we generalize the basic 1-to-1of2 solution to 1-to-1ofN solutions, for N>2.

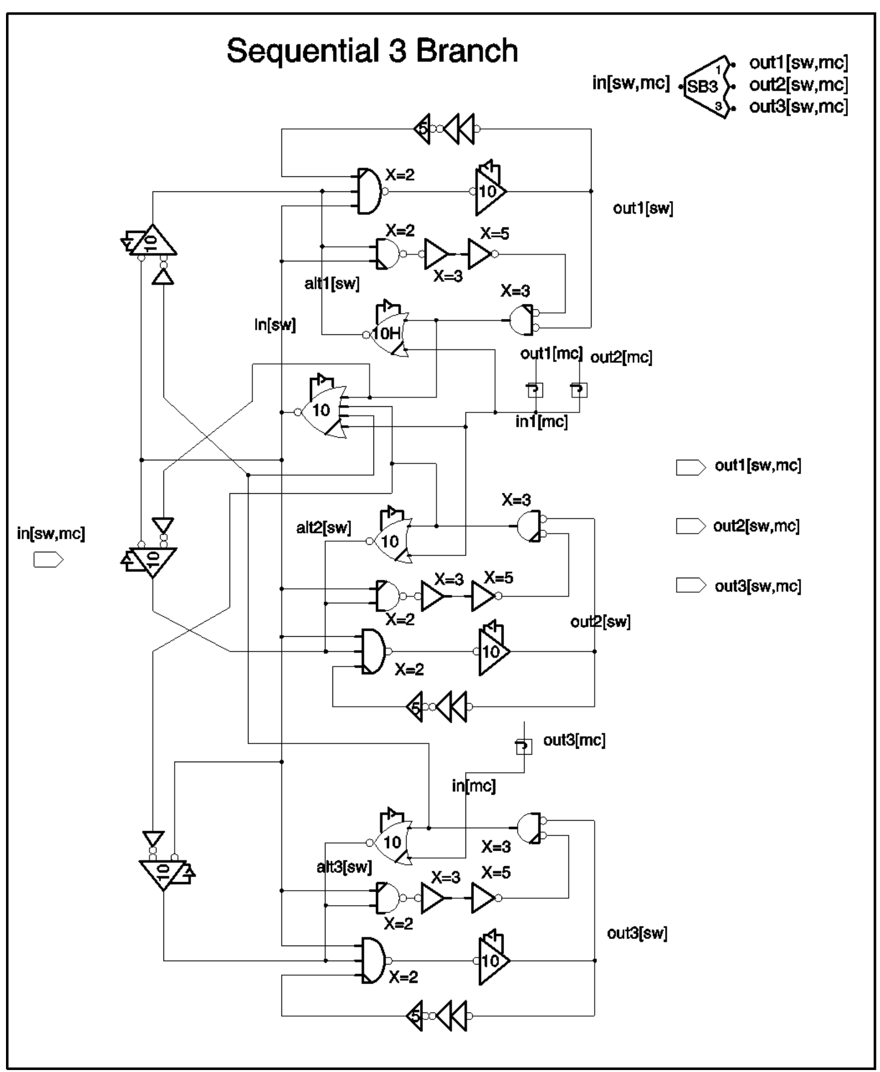


Figure 36: Telescope GasP implementation for the 3-way Sequential Branch module. The circuit is a straightforward extension of the Toggle Branch circuit in **Figure 31**. The token ring now contains three statewires: **alt1[sw]**, **alt2[sw]**, and **alt3[sw]**. The three statewires are connected so that the single HI "token" goes around in a round-robin fashion. The 3-way Sequential Branch module has one input statewire with a master clear signal, **in[sw,mc]**, and three output statewires with a forwarded master clear signal, **out1[sw,mc]**, **out2[sw]**, and **out3[sw,mc]**. We will use the top-right icon to represent a 3-way Sequential Branch.

5.3. 4-Way Sequential Branch Module

The 4-way Sequential Branch module provides single input to single output communication. Whenever it receives a communication over its input channel, **in[sw]**, it streams the corresponding data to one of the output communication channels, **out1[sw]**, **out2[sw]**, **out3[sw]**, or **out4[sw]**. It selects the output channel in a round-robin fashion, starting with **out1[sw]**.

Figure 37 and **Figure 38** give our Telescope GasP implementation for the 4-way Sequential Branch module, and a SPICE level simulation showing its sequential input behavior.

Rather than extending the circuit implementations of the 2-way Sequential Branch modules in **Figure 31** and **Figure 32** for the first and second design approaches, we designed a balanced tree structure consisting 2-way Sequential Branch modules. We did this design in Electric [14], and provided it with its own icon. But we could as easily do this design in ARCwelder [6], except that ARCwelder does not yet support hierarchical designs and naming and icon conventions for sub-designs.

Without a fast reset signal, the minimum cycle time of this design is 2+2+5+2+2+5=18 gate delays. If we were to add a fast reset signal, the cycle time would decrease to 14 gate delays.

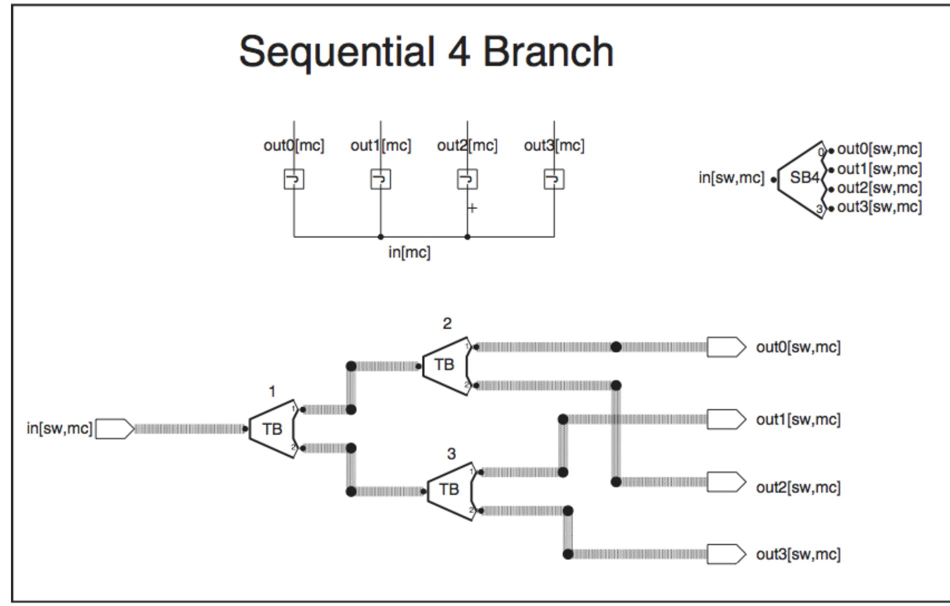


Figure 37: Telescope GasP design solution for a 4-way Sequential Branch. The 4-way Sequential Branch module has one input statewire with a master clear signal, **in[sw,mc]**, and four output statewires with a forwarded master clear signal, **out1[sw,mc]**, **out2[sw]**, **out3[sw]**, and **out4[sw,mc]**. This is a hierarchical design, built using three instances of a 2-way Sequential Branch or Toggle Branch module. We connected the instances to form a balanced tree structure. The tree structure comes with a penalty of larger latencies and a larger cycle time. It induces a total forward latency and total backward latency of 4 gate delays each: 2 gate delays more in each direction than for a basic Telescope GasP module. A fast reset signal would cut the backward latency down to 0 gate delays. We will use the top-right icon to represent a 4-way Sequential Branch.

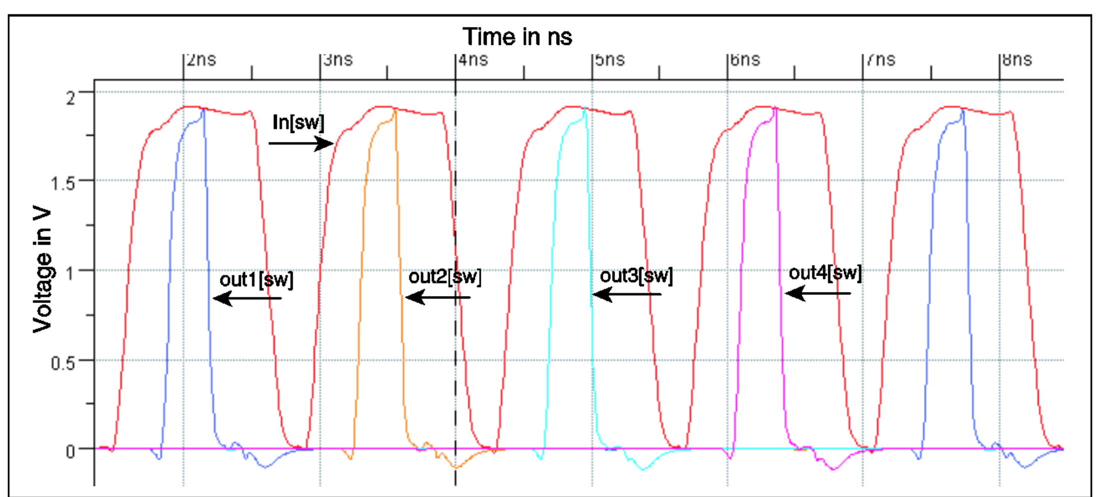


Figure 38: SPICE-level simulations of the GasP implementation of the 4-way Sequential Branch in

Figure 37, using the 2-way Sequential Branch or Toggle Branch implementation of **Figure 31**. The telescoping handshake behavior would be the same had we used the 2-way Sequential Branch implementation of **Figure 32**. The simulation clearly shows that (1) each HI output pulse is strictly contained within a HI pulse on in[sw], and that (2) in[sw] handshakes go in a round-robin fashion with out1[sw], out2[sw], out3[sw] and out4[sw] handshakes.

6. Distribute Modules

The Distribute modules stream one input to a possibly empty subset of N outputs. The subset can be tuned per communication and is set via N additional incoming control bits. As in the Branch modules, we opted to broadcast the input data to all outputs, no matter whether or not the receiving module participates in the action. As a result, no special steering logic is needed. Sections 6.1, 6.2, and 6.3 below give Telescope GasP designs for respectively 2-way, 3-way and 4-way Distribute modules.

6.1. 2-Way Distribute Module

The 2-way Distribute module streams the data from one incoming communication channel, **in[sw]**, to a subset of 2 outgoing communication channels, **out1[sw]** and **out2[sw]**. The selection of outputs to stream the data to depends on 2 additional incoming "guard" or control data bits **d[1:2].** The control data bits are bundled with an incoming control statewire, called **to[sw]** in **Figure 39** and called **guard[sw]** in **Figure 40**. Channel **outi[sw]** is in the selection if its controlling data bit **d[i]** is HI, for i ranging from 1 to 2.

Figure 39 and **Figure 40** show our two Telescope GasP implementations. The version in **Figure 39** is an example of our first design approach with independent self-resetting loops for statewires of incoming and outgoing handshake channels. It has two self-resetting loops to start and stop the drives for external, incoming or outgoing, statewires. The top loop has two taps to start and stop the HI drive for *out1[sw]* and *out2[sw].* The bottom loop has two taps to start and stop the LO drive for *in[sw]* and *to[sw].* The implementation in **Figure 32** is an example of the second design approach with shared self-resetting loops between the statewires of incoming and outgoing handshake channels. It also has two self-resetting loops for external statewires, with separate taps to start and stop the HI drives for *out1[sw]* and *out2[sw]* and the LO drive for *in[sw].*

SPICE-simulated waveforms of the handshake behavior of the implementation in **Figure 39** follow in **Figure 41**. The handshake behaviors are the same for both implementations, and can be described as follows. Initially, **in[sw]**, **to[sw]**, **out1[sw]**, **out2[sw]**, and master clear signal **mc** are LO. When **in[sw]** goes HI, indicating it has data, and **to[sw]** goes high, indicating that the control data bits **d[1:2]** have been updated, and both outputs **out1[sw]** are LO, indicating there is enough space to receive the incoming data, then the following four actions are started **sequentially** in the order indicated:

1. The Forward Transfer part signals the presence of new data to the successor modules of the selected outputs by driving these outputs HI. This takes 2 gate delays.
2. The Forward Reset section kicks in, cutting off the HI forward drives to a 5 gate delay pulse.
3. The Backward Transfer waits until the selected outputs are LO — which may be directly following action 1 above, in case no outputs were selected — and then it reports availability of space to the predecessor by driving **in[sw]** and **to[sw]** LO. In case at least one output was selected, **in[sw]** and **to[sw]** are driven LO 2 gate delays after the last selected output goes LO. If no output was selected, **in[sw]** and **to[sw]** are driven LO 5 gate delays after **in[sw]** went HI. The latter case has an additional waiting time of 3 gate delays to accommodate a 5 gate delay HI pulse on **in[sw]**.
4. The Backward Reset section kicks in, cutting off the LO backward drives to a 5 gate delay pulse.

At the end of these four **serial** actions, the 2-way Distribute module waits until inputs **in[sw]** and **to[sw]** are HI again, so it can start its next cycle.

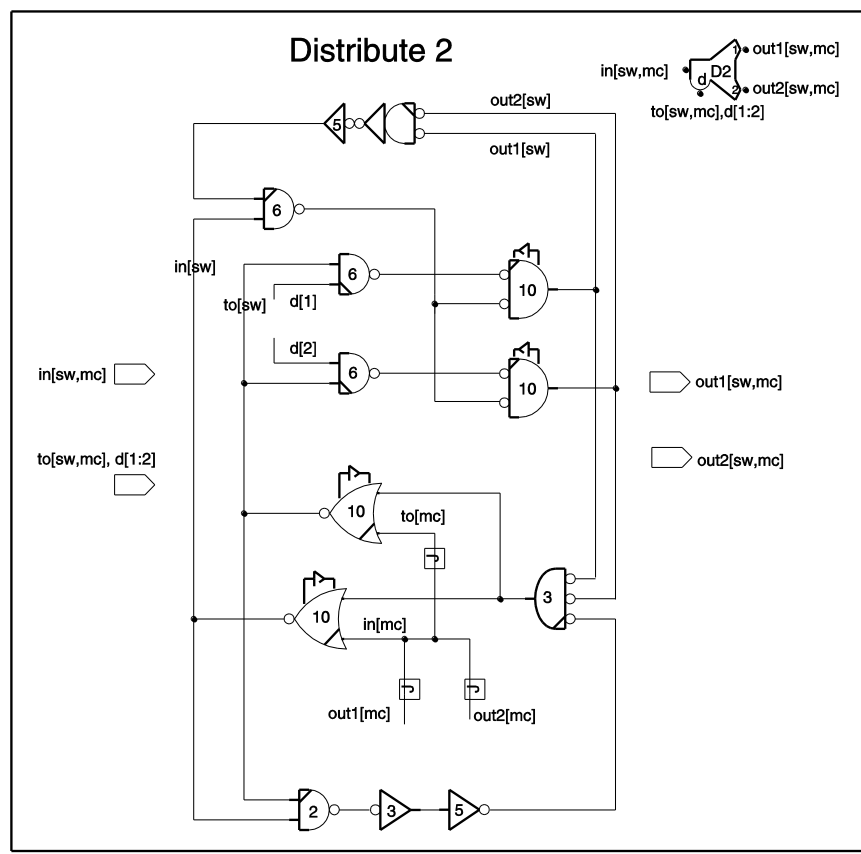


Figure 39: Telescope GasP implementation for the 2-way Distribute module. It has two input statewires with a master clear signal, **in[sw,mc]** and **to[sw,mc]**, and two output statewires with a joined master clear signal, **out1[sw,mc]** and **out2[sw,mc]**. The two data control signals **d[1:2]** have a bundled-delay relation to input statewire **to[sw,mc]**. We will use the icon in the top-right corner to represent a 2-way Distribute module. This particular implementation is saved under the library TelescopeGasP_SeparateStates to indicate that it uses separate self-resetting loops for statewires of incoming versus outgoing handshake channel**.**

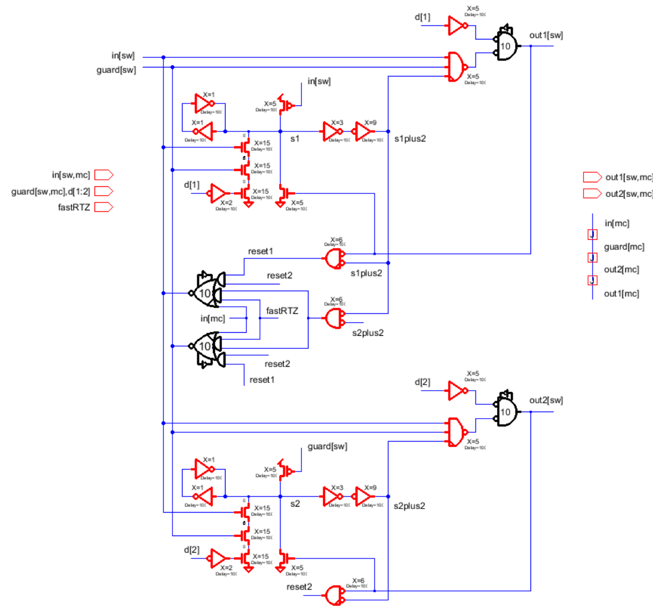


Figure 40: Second Telescope GasP implementation for a 2-way Distribute module, different from **Figure 39**. Here, we share the self-resetting loops between incoming and outgoing statewires. This implementation is saved in the library named TelescopeGasP_wStateSharing, under module name Distribute2TfastRTZ. Substring `TfastRTZ' indicates that this is a Telescope GasP implementation with fast reset capability.

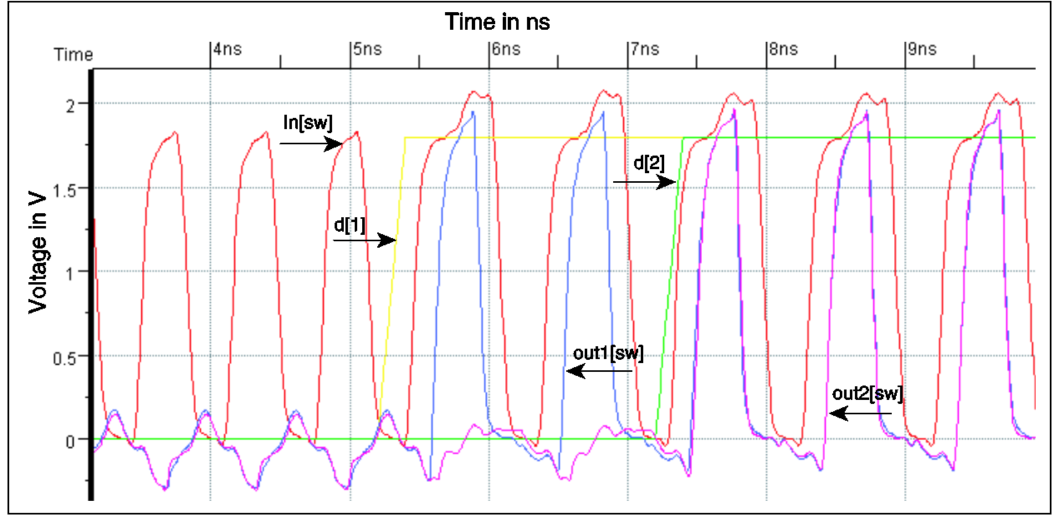


Figure 41: SPICE-level simulations of the GasP implementation of the 2-way Distribute in **Figure 39**. Red statewire in[sw] has a telescopic handshake relationship to blue and pink statewires ou1t[sw] and out2[sw]: in[sw] HI causes ou1t[sw] to go HI if yellow signal d[1] is HI, and it causes out2[sw] to go HI if green signal d[2] is HI. The yellow and green data control signals are bundled with signal to[sw] not shown here. When the respective output have gone LO, only then will in[sw] go LO. The simulation shows telescope handshake cycles with 0, 1 and 2 participating outputs, and demonstrates that (1) each HI output pulse is strictly contained within a HI pulse on in[sw], and (2) outi[sw] participates if and only if d[i] is HI, for i=1,2.

Internally, the behaviors of the two implementations differ. We would like to emphasize three differences:

1. For one, the implementation in **Figure 40** has relatively safe delay margins, whereas the implementation in **Figure 39** has two delay margins that are marginal. The two marginal delay margins in **Figure 39** are as follows:
   - After *in[sw]* and *to[sw]* have risen, it takes 2 gate delays before the chosen signals in the subset of *out1[sw]* and *out2[sw]* rise and disable the inverted input AND gate in the bottom loop, while it takes 3 gate delays for the two incoming signals to enable the same inverted input AND gate via the bottom self-resetting loop. To guarantee a telescope relation from *in[sw]* and *to[sw]* to *out1[sw]* and *out2[sw]*, the AND gate must sense the disabling transitions before it senses the enabling one. The delay margin between the two is only 1 gate delay.
   - After the chosen signals in the subset of *out1[sw]* and *out2[sw*] fall, it takes 2 gate delays before *in[sw]* falls and disables the two input NAND gate in the top loop, while it takes 3 gate delays for the chosen outgoing signals to enable this NAND gate via the top self-resetting loop. For a correct handshake relation, the NAND gate must sense the disabling transition before it senses the enabling one. The delay margin between the two is only1 gate delay.

   A delay margin of only 1 gate delay is marginal and sensitive to noise interference and process variations. We can adjust both margins to 3 gate delays by inserting the proper delay circuits from **Figure 1**, page 3. This would make the delay margins in the implementation of the 2-way Distribute in **Figure 39** as safe as those in **Figure 40**.
2. Second, the first design approach used in **Figure 39** has the advantage that its control-data bundling constraints are less strict than those for the second design approach used in **Figure 40**:
   - Statewire *to[sw]* in **Figure 39** comes in at the same gates as its bundled data bits *d[1]* and *d[2].* Thus, the control-data bundling constraints between *to[sw]* and *d[1:2]* remain the same as they were prior to entering the Distribute module.
   - In contrast, statewire *guard[sw]* in **Figure 40** arrives one inverter gate earlier than its bundled control bits *d[1]* and *d[2]* arrive at the PMOS stacks to pull down *s1* and *s2* to help reset *in[sw]* and *guard[sw]* in case any of the outgoing channels do not participate in the action. Consequently, the control-data bundling constraints between *guard[sw]* and *d[1:2]* change: the Distribute module makes it harder for the control to arrive before the data than it was prior to entering the module. It makes it harder by a margin of 1 gate delay.
3. Third, unlike earlier module designs in this document and in [9], this time the first design approach leads to a simpler implementation than the second design approach:
   - For one, the second implementation has special logic to reset the incoming channels in case any of the outgoing channels does not participate. The first implementation seems to have no need for this. This is because the first implementation in **Figure 39** has yet to be corrected for marginal delay margins, as explained in item 1 above. As soon as we add the proper delay circuits from **Figure 1** on page 3, the need for *d[1:2]* for resetting the incoming channels will become obvious.
     - The bottom loop in **Figure 39** is responsible for resetting the incoming channels. It needs a delay circuit of the type shown in **Figure 1**(right) to add two extra gate delays to the AND-ed transition in the bottom loop that starts with *in[sw]* and *to[w]* both HI and ends with the corresponding input for the inverted input AND gate going LO.
     - With the delay circuit from **Figure 1**(right), but without *d[1:2],* this HI to LO bottom loop transition will take 5 gate delays, always. In particular, it will take 5 gate delays when *in[sw]* and *to[sw]* go HI while *d[1]* and *d[2]* are LO, and 2 more to reset *in[sw]* and *to[sw]*, giving a total HI to LO cycle time for *in[sw]* and *to[sw]* of 7 gate delays when no data are distributed. Compare this to the implementation in **Figure 40** which does this in 5 gate delays.
     - There is a simple solution for reducing the HI to LO cycle time in the absence of data from 7 to 5 gate delays: only delay the HI to LO bottom loop transition when *d[1]* and *d[2]* are LO. We can do this by adjusting the delay circuit of **Figure 1**(right) as shown in **Figure 42**(bottom).
   - The resulting circuit for **Figure 39** with top and bottom gates replaced by the circuits shown in **Figure 42** is still simpler than the 2-way Distribute implementation for the second design approach shown in **Figure 40**. This is because the second approach focuses on individual input-to-output connections. When no data is distributed, these individual connections are bypassed and the bypasses are gathered to jointly reset the inputs. The individual bypass and gather circuitry in **Figure 40** is more complex than the joined *d[1:2]* bypass in the delay circuits for **Figure 39**.

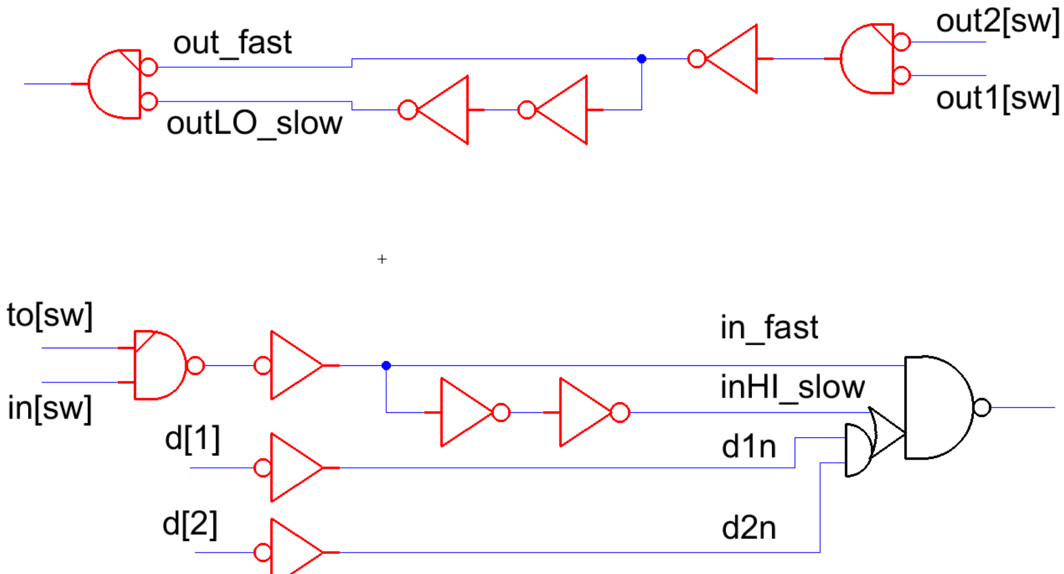


Figure 42: Delay circuits replacing the top three gates in **Figure 39** (top) and the bottom three gates (bottom). Both delay circuits are variants of the earlier delay circuits shown in **Figure 1**. The top circuit increases the delay margin from **out2[sw]** and **out1[sw]** LO to the arrival of the corresponding HI transition at the 2-input NAND gate in the top loop. The margin increases by 2 gate delays. The bottom circuit gives a similar increase in delay margin from **in[sw]** and **to[sw]** HI to the arrival the corresponding LO transition at the 3 inverted input AND gate in the bottom loop, as long as either **d[1]** or **d[2]** is HI. The final gate in the bottom circuit is implemented directly as a PMOS-NMOS transistor and channel connected network, and takes 1 gate delay

The two implementations have the same cycle time and the same latencies. The two Forward Transfers in **Figure 39** and **Figure 40** use the Succ AND Driver in [9]. There is 1 extra gate between **in[sw]** respectively **to[sw]** and the inputs of the two Succ AND Drivers, giving a minimal forward latency of 2 gate delays. The two Backward Transfer parts are implemented using a Pred Driver and a Pred AND Driver. The Pred AND Driver is used in **Figure 40** and has an additional fast reset – see **Figure 10**. In both cases, there is 1 additional gate between the outgoing channel **out1[sw]** and **out2[sw]** and the input of the corresponding Pred OR Driver or Pred AND Driver, giving a total backward latency of 2 gate delays.

Without a fast reset signal, both implementations have a minimum cycle time of 2+5+2+5=14 gate delays, as set by the 2 gate delay transfers and the 5 gate delay self-resetting loops - assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of 4 gate delays for each Telescope GasP successor module between the Distribute module and the following Store module.

With a fast reset signal, we can reduce the backward latency and lower the cycle time to 12 gate delays per 2-way Distribute module, with an increase in steps of 2 gate delays for each successor module between the current Distribute module and the following Store module. We designed a test environment that generates such a fast reset signal. The test setup and the generation of the fast reset signal, fastRTZ, are very similar to the test setup and fast reset generation for the Merge and Branch modules. Test setup and simulation waveforms for the 2-way Distribute with fast reset follow in **Figure 43** and **Figure 44** below.

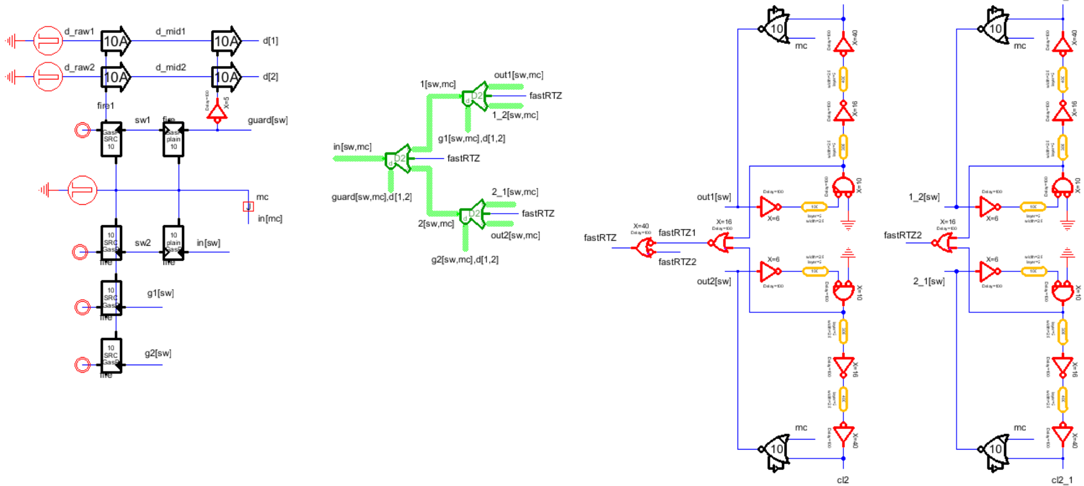


Figure 43: Fast reset configuration and test environment to concurrently reset all Distribute statewires that actively participate in the current handshake telescope leading to **out1[sw]** or **out2[sw]** or **1_2[sw]** or **2_1[sw]** going HI. We use the 2-way Distribute implementation of **Figure 40**. Each Store module at the right generates a local fast reset signal for the Distribute module that it is connected to. These local fast reset signals are OR-ed to produce the common fastRTZ signal that resets all participating Distribute statewires concurrently. This solution is less general than we would like it to be. For one, it requires that the Distribute tree is balanced. We anticipate that, in practice, we will be able to identify a particular set of Store modules from which to generate such a common fastRTZ signal.

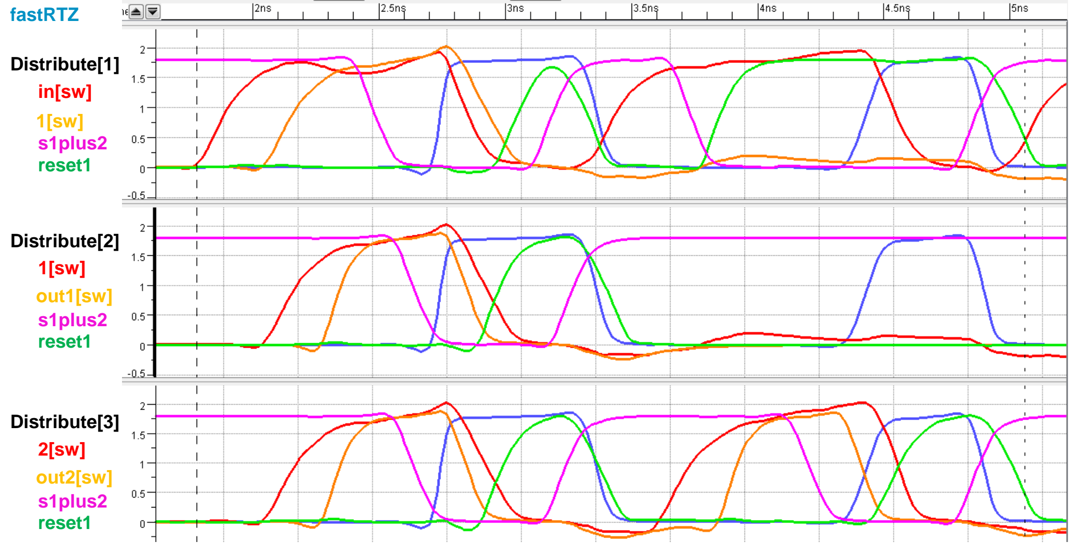


Figure 44: SPICE simulation results for two telescope handshakes in the fast reset and test configuration in **Figure 43**. During the 5 gate delay HI pulse on fastRTZ, all currently HI orange and red handshake signals are reset to LO, simultaneously. Just like was the case for the Merge and Branch modules, the differences in start and steepness of the falling slopes for the statewires of each Distribute module can be explained from differences in (a) input slopes to the LO drivers for the statewires and (b) stray capacitance on the statewires.

6.2. 3-Way Distribute Module

The 3-way Distribute module streams the data from one incoming communication channel, **in[sw]**, to a subset of 3 outgoing communication channels, **out1[sw]**, **out2[sw]**, and **out3[sw]**. The selection of outputs to stream the data to depends on 3 additional incoming "guard" or control data bits **d[1:3].** The control data bits are bundled with an incoming control statewire, called either **to[sw]** or **guard[sw]**. Channel **outi[sw]** is in the selection if its controlling data bit **d[i]** is HI, for i ranging from 1 to 3.

The generalization from the two-way Distribute implementation in **Figure 39** to the three-way version in **Figure 45** has a price – though the price that we paid here is much more reasonable than the price that we paid for the two-way to three-way generalization of the Sequential Branch module. There are still only two self-resetting loops to start and stop the drives for statewires of external, incoming or outgoing handshake channels. But instead of Pred Drivers the 3-way version uses Pred AND Drivers to drive **in[sw]** and **to[sw]** LO – see [9] for details on the two predecessor driver designs.

Generalizing the fast reset version for the second design in **Figure 40** is more challenging: the predecessor drivers would be replaced by a Pred AND3 Driver with fast reset. Although do-able, we are hesitant to keep on making the drivers more and more complex.

There are other ways to build N-way Distribute modules for N larger than 2:

- Instead of adding functional complexity to a single module, we could distribute the complexity over multiple modules, like we do for the four-way Distribute in the subsequent Section 6.3.
- We could relax the 2 gate delay forward and backward latencies in more complex N-way module designs, where N exceeds 2. This would enable us to spread the complexity over multiple gates and drivers that are relatively simple. A module with a forward and backward latency of 3 or 4 gate delays would still have a smaller or equal latency compared to a design with two or more modules in series.

Note:

- We do not provide a behavioral description and waveform for the 3-way Distribute module. The behavior is a straightforward extension of the behavior for the 2-way version. The main motivation for designing and presenting this 3-way version is to get and give a feel for the growth in design complexity when we generalize the basic 1-to-1to2of2 solution to 1-to-1toNofN solutions, for N>2.

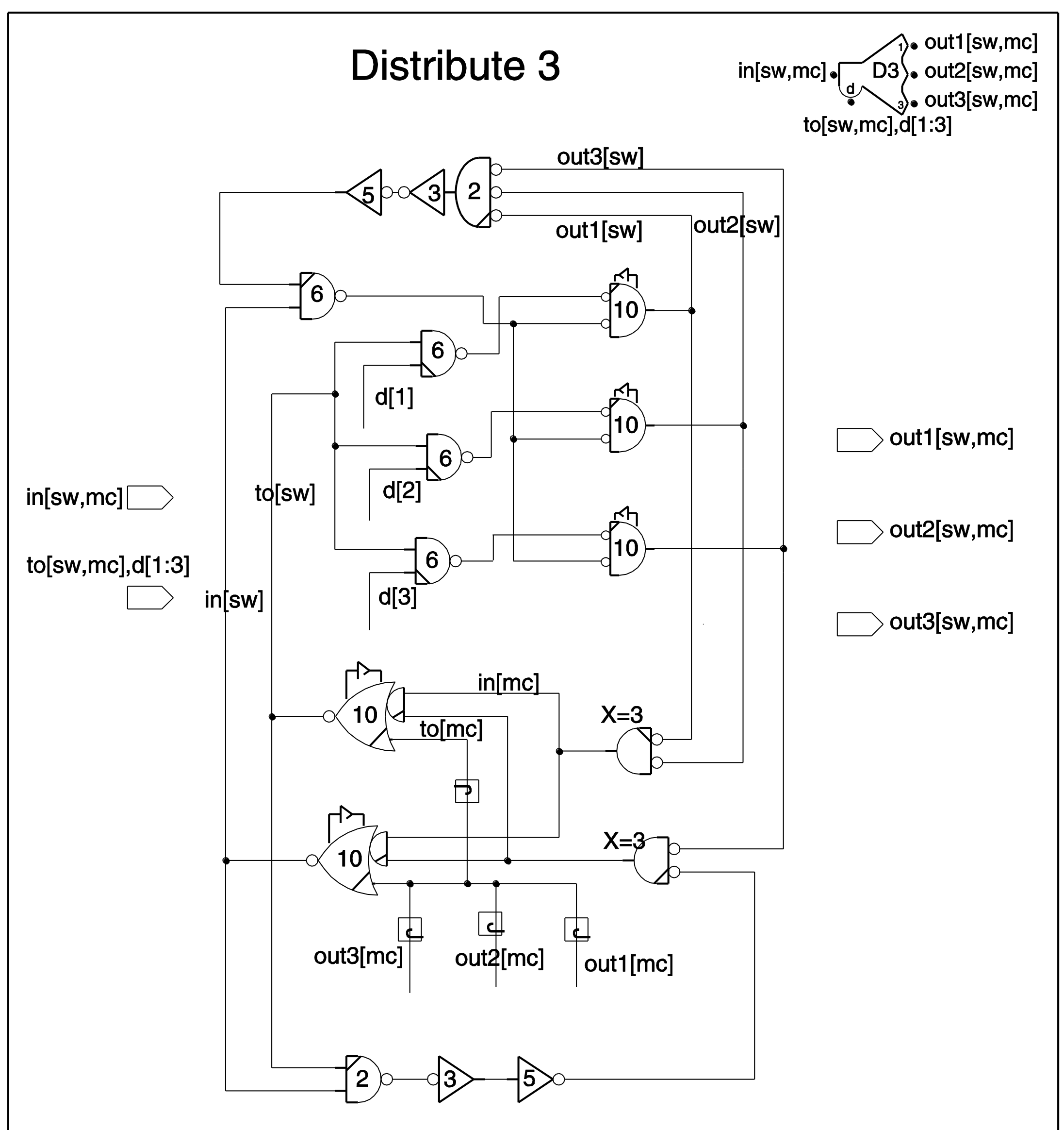


Figure 45: Telescope GasP control circuit for the 3-way Distribute module. This design is a generalization of the 2-way Distribute module in **Figure 39**. It has two input statewires with a master clear signal, **in[sw,mc]** and **to[sw,mc]**, three output statewires with a joined master clear signal, **out1[sw,mc]**, **out2[sw,mc]**, and **out3[sw,mc]**. Data control signals **d[1:3]** have a bundled-delay relation to statewire **to[sw,mc]**. We will use the icon in the top-right corner to represent a 3-way Distribute module.

6.3. 4-Way Distribute Module

The 4-way Distribute module streams the data from one incoming communication channel, **in[sw]**, to a subset of 4 outgoing communication channels, **out1[sw]** ,**out2[sw]**, **out3[sw]**, and **out4[sw]**. The selection of outputs to stream the data to depends on 4 additional incoming control data bits **d[1:4].** The control data bits are bundled with an incoming control statewire, called either **to[sw]** or **guard[sw]**. Channel **outi[sw]** is in the selection if its controlling data bit **d[i]** is HI, for i ranging from 1 to 4.

Figure 46 and **Figure 47** give our Telescope GasP implementation for the 4-way Distribute module, and a SPICE level simulation showing its behavior.

Rather than extending the circuit implementation of the Distribute 2 and Distribute 3 modules in **Figure 39** and **Figure 45**, we partitioned the design into two layers of Telescope GasP modules: two 2-way Fork modules from [9] followed by two 2-way Distribute modules. One of the two-way Forks broadcasts the input data to both Distribute modules, and the other one partitions the control data over the Distribute modules. They do this concurrently. The 2-way Distribute modules each control two output channels. Between them, they cover the data-controlled distribution over all four output channels.

We did this design in Electric [14], and provided it with its own icon. We could, in principle, have done the design in ARCwelder [6], except that ARCwelder does not yet support hierarchical designs and naming and icon conventions for sub-designs.

The minimum cycle time of this design is 2+2+5+2+2+5=18 gate delays. If we were to add a fast reset signal, the cycle time would decrease to 14 gate delays.

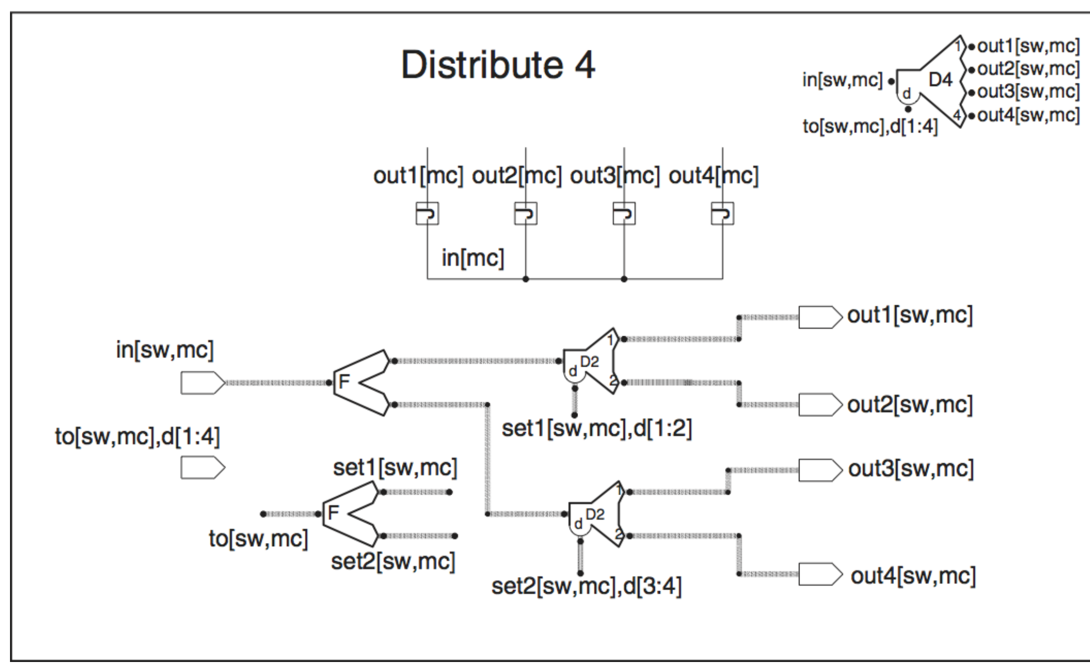


Figure 46: Telescope GasP design of a 4-way Distribute. It has two input statewires with a master clear signal, **in[sw,mc]** and **to[sw,mc]**, and four output statewires with joined master clear signal, **out1[sw,mc]**, **out2[sw]**, **out3[sw]**, and **out4[sw,mc]**. Data control signals **d[1:4]** have a bundled-delay relation to input statewire **to[sw,mc]**. This is a hierarchical design, built using two instances of the 2-way Fork module in [9] and two instances of the 2-way Distribute module in **Figure 39**. One Fork broadcasts the data on **in[sw]** to both Distribute modules, the other partitions the control data over both Distribute modules. They do this concurrently. The 2-way Distribute modules each control two output channels. Between them, they cover the data-controlled distribution over all four output channels. We will use the icon in the top-right corner to represent this 4-way Distribute design.

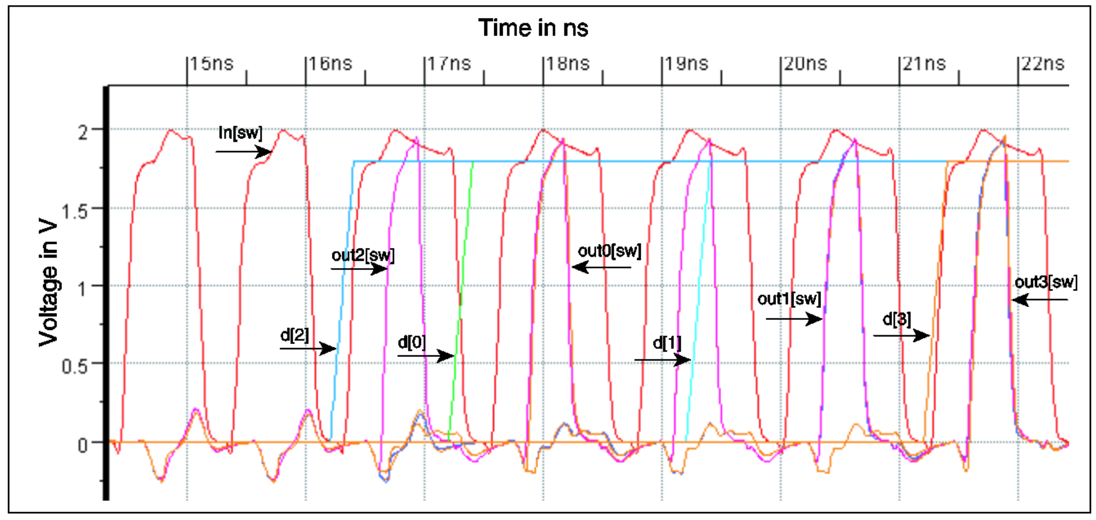


Figure 47: SPICE-level simulations of the 4-way Distribute of **Figure 46**, showing that (1) each HI output pulse is strictly contained within a HI in[sw] pulse in red, and (2) outi[sw] participates if and only if d[i] is HI, for i=1,2.

7. Conclusion and Future Work

This document gives the implementation and simulation details of Telescope GasP modules for narrowcast and arbitrated or data driven control used in ARCwelder [5,6]: Merge, Branch, and Distribute.

Telescope GasP implementations have a minimum cumulative cycle time of 10+4k gate delays, where k is the number of consecutive Telescope GasP modules in the data path following and including the present module up the Store modules that store the computed results at the end of the data path. The increase in cycle time of 4 gate delays per data path module is due to the use of single-track channels. This puts single-track designs styles, like Telescope GasP, at a disadvantage over other telescoping handshake design styles that use multi-track handshake signaling, such as Click [13]. We can alleviate this disadvantage and obtain a cumulative cycle time of 10+2k gate delays by adding a fast reset strategy.

The fast reset solutions presented in this document share the following key features:

1. The fast reset signal is generated from the final Store module in the data path. We anticipate that, in case of the Distribute module, we will be able to identify a particular set of Store modules from which we can tap a fast reset signal. We leave such identification for future work.
2. Unlike the fast reset solutions for the broadcast modules in [9] where we re-used the existing master clear input signal to the module as fast reset signal input, the modules in this document use a separate fast reset input that function according to the logical effort expectations of a normal post-initialized Telescope GasP operation.
3. By convention, statewires in Telescope GasP are initialized LO. So, the separate fast reset input will be an extra input for the predecessor driver. The modules in this document select which incoming and outgoing handshake channels will participate in the current telescope action. To ensure that only the incoming handshakes that participate are reset, we AND the fast reset signal with an internal guard signal. The guard indicates whether or not to reset the handshake channel.
4. This fast reset strategy adds two extra inputs to the predecessor drivers. This has a noticeable impact on the complexity of the circuitry for predecessor drivers. The driver complexity also makes it harder to generalize the module designs for more incoming or outgoing channels, as we observe in Section 5.2, where we generalize the 2-way Sequential Branch to a 3-way version.

To generalize Merge, Branch, and Distribute modules for more incoming and outgoing channels, we can:

- Use more complex gates and drivers.
- Spread the function over multiple modules, like we do for the four-way Distribute in Section 6.3.
- Increase the 2 gate delay forward and backward latencies to 3 or 4 gate delays, by spreading the action over more and simpler gates and drivers. A module with a forward and backward latency of 3 to 4 gate delays still has a smaller or equal latency compared to two or more modules in series.

Unlike the broadcast modules in [9], the modules in this document steer data selectively. We forward only those data bits that are bundled with incoming handshake channels that participate in the current telescope action. We opted to broadcast the incoming data to all outgoing handshake channels, no matter whether or not the receiving module participates in the action. As a result, no special steering logic is needed for the single-input, multiple-output Branch and Distribute modules. But Merge modules *do* need special steering logic. Each of our Merge modules has a data steering signal per incoming handshake channel. The signal is HI when the data bundled with the handshake channel are to be forwarded, and it is LO when the bundled data are to be ignored. Such steering signals can be used to control the multiplexers in the data path that multiplex the data bits coming into the module onto the module's outgoing data bits.

As before, we used two design approaches to implement the modules in this document. Both approaches have their advantages and disadvantages. The advantage of the first approach is that its control-data bundling constraints are less strict than those for the second approach, as we demonstrate for the 2-way Distribute module in Section 6.1. The advantage of the second approach is that it has better delay margins than the first approach; this is true for all modules discussed here.

Appendix C

Telescope GasP: Repeat

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapters 2 and 3 and can be found as reference [27] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Telescope GasP: Repeat. *Technical Report, ARC2012-smg04, Asynchronous Research Center, Portland State University*, March, 2012.

Asynchronous Research Center

Portland State University

Subject: **Telescope GasP: Repeat**
Date: **1 March, 2012**
From: **Swetha Mettala Gilla, Marly Roncken and Ivan Sutherland**
ARC#: **2012-smg04**

References:

[1] Peter Beerel, Georgios Dimou, and Andrew Lines. Proteus: an ASIC Flow for GHz Asynchronous Designs, *IEEE Design & Test of Computers, Vol. 28, Issue 5*, pp. 36-51, 2011.
[2] Andrew Bardsley, Luis Tarazona, and Doug Edwards. Teak: A Token-Flow Implementation for the Balsa Language, In *Proceedings of the International Conference on the Application of Concurrency to System Design (ACSD)*, pp. 23-31, 2009.
[3] Teak asynchronous synthesis, Univ. of Manchester: http://apt.cs.man.ac.uk/projects/teak/.
[4] Jo Ebergen, Bill Coates, and Austin Lee. Long-Distance On-Chip Communication using GasP. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 119-116, 2011.
[5] Willem Mallon and Ivan Sutherland. Icons for Click and GasP Modules, *ARC2011-is05, Technical Report, Asynchronous Research Center, Portland State University,* March 2011.
[6] Asynchronous Research Center website: http://arc.cecs.pdx.edu/.
[7] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Long-Range GasP with Charge Relaxation. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 185-195, 2010.
[8] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Telescope GasP: Overview. *ARC2012-smg01, Technical Report, Asynchronous Research Center, Portland State University,* 2012. **Note: ARC2012-smg01 has been fully integrated into Chapter 2 of Swetha's Ph.D. thesis.**
[9] — GasP Storage and Telescope GasP Broadcast: Store, Amplify, Fork, and Join. *ARC2012-smg02.* **Note: ARC2012-smg02 is included as Appendix A in Swetha's Ph.D. thesis.**
[10] — Telescope GasP Narrowcast and Arbitrated or Data Driven Control: Merge, Branch, and Distribute. *ARC2012-smg03.* **Note: ARC2012-smg03 is included as Appendix B in Swetha's Ph.D. thesis.**
[11] — Telescope GasP: Repeat. *ARC2012-smg04.* **Note: ARC2012-smg04 is included as Appendix C in Swetha's Ph.D. thesis.**
[12] — Arbiter Design Improvements for GasP. *Technical Report, ARC2012-smg05, Work in Progress.* **Note: ARC2012-smg05 is now finished and included as Appendix G in Swetha's Ph.D. thesis.**
[13] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An Implementation Style for Data-Driven Compilation. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 3-14, 2010.
[14] Steven M. Rubin. Using the ELECTRIC™ VLSI Design System, Version 8.11. *Static Free Software and Sun Microsystems, ISBN 0-9727514-3-2,* R.L. Ranch Press, 2010.
[15] Charles Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems, Carver Mead and Lynn Conway (Eds), Addison-Wesley*, pages 218-262, 1980.
[16] Ivan Sutherland. A Mutual Exclusion Pass Gate, *ARC2010-is26, Technical Report, Asynchronous Research Center, Portland State University,* May 2010.
[17] Ivan Sutherland. Fourth Class Handout: Proper Stopper, *ARC2012-is49, Technical Report Asynchronous Research Center, Portland State University,* October 2010.
[18] Ivan Sutherland and Scott Fairbanks. GasP: A Minimal FIFO Control. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 46–53, 2001.
[19] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow, In *Proc. International Workshop on Logic Synthesis, 2004.*

This document contains information developed at the Asynchronous Research Center at Portland State University. Disclose this information to whomever you choose. Distribution is permitted with adequate reference to the source of the ideas and information. You may reproduce this material for any educational use. Copies of the material must contain this notice.

Abstract
This document gives Telescope GasP implementations for Repeat modules used in ARCwelder [5,6] to implement repetitive data streaming. ARCwelder, previously called TPDesign, is a design and compilation environment for dataflow computations; we use it to design self-timed systems. Its organization builds upon results and key learnings of the data-driven compiler effort by Handshake Solutions [13]. Related self-timed compilation efforts can be found in for instance [1,2,3,19].

An overall motivation for Telescope GasP, including pros, cons, and alternatives, can be found in [8]. Implementations for storage and broadcast modules are given in [9], Module implementations for narrowcast 1-to-1ofN or 1ofN-to-1, or for 1-to-1toNofN communications are given in [10]. This document [11] closes the present series of Telescope GasP modules by giving implementations for Repeat modules. The references in the current ARC report list all references on Telescope GasP used here or in [8,9,10,11].

Keywords: GasP backend for ARCwelder (TPDesign), Telescope GasP for repetitive data streaming.

Notation:

1. All GasP implementations in this document and in the related documents [8,9,10] are non-inverting, i.e., their handshake channels, a.k.a. statewires, use the same encoding. Statewires say whether the data bundled with the statewire are valid. We use:
   - HI for a high voltage level, e.g. VDD, to indicate a handshake REQUEST phase with valid data.
   - LO for a low voltage level, e.g. VSS or GND, to indicate a handshake ACKNOWLEDGE phase where data are no longer needed.

   In our circuit diagrams, we use a short 45-degree line segment for VDD and a triangle for VSS. A PMOS transistor connected to VDD and an NMOS transistor connected to VSS are depicted as:

2. Our schematic designs use a so-called JBOX construct. This is a special construct in Electric [14] to connect signals with different names. The symbol for a JBOX construct is a box with the letter J in it. We specifically use it to join the **master clear (mc)** signals on the various statewires of a module. For example, **Figure 2** on page 4 uses a JBOX to connect master clear signals in[mc] and out[mc].

Acknowledgements: We gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD thesis work.

Table of Contents

References: 1
1. Introduction 3
2. Custom Driver Designs for Repeat Modules 4
   2.1. Single-Track Statewire Drivers for Pull-HI and Keep-LO 4
   2.2. Single-Track Statewire Drivers for Pull-LO and Keep-HI 5
3. Repeat Module 6
   3.1. RepeatWhile 6
   3.2. RepeatUntil 12
4. Conclusion and Future Work 18

1. Introduction

This document gives GasP control implementations for Repeat modules. These modules are used in ARCwelder [5,6] to implement repetitive data streaming. Below follows a short description:

- **Repeat**
  - We implemented two types of **Repeat** module, only one of which was listed in [5]. We dubbed the listed version **RepeatUntil**, and added **RepeatWhile**.
  - **RepeatUntil** streams the input data to the output at least once. After each output, it waits for a guard input to determine whether or not to stream the same data to the output once more.
  - **RepeatWhile** streams the input data to the output zero or more times. Initially, and again after each output, it waits for a guard input to determine whether or not to stream the input data to the output one more time. In GasP, the RepeatWhile module is easier to implement than the RepeatUntil module, and therefore, we will present its implementation details first.
  - Implementation details follow in Section 3.

When designing GasP modules, we distinguish and design four communication control actions:

1. **Forward Transfer**
   The HI handshake request signal coming in from the predecessor statewire transfers into a HI request signal on the successor statewire. The actual transfer generally requires synchronization with other incoming and with outgoing signals. We opted for a forward transfer latency of 6 gate delays for the Store module and of 2 gate delays for the other modules.
2. **Backward Transfer**
   The LO handshake acknowledge signal coming in from the successor statewire transfers into a LO acknowledge signal on the predecessor statewire. The transfer generally requires synchronization with other outgoing and with incoming signals. We opted for a backward transfer latency of 4 gate delays for the Store module and of 2 gate delays for the other modules.
3. **Forward Reset**
   The forwarded HI request signal on the successor statewire turns off its own HI drive after 5 gate delays, thereafter relying on the keeper to keep the statewire HI as long as needed. The Forward Reset in a storage-free Telescope GasP module strictly precedes its Backward Reset. Traditional GasP modules, like module Store, turn off their forward HI and backward LO drives concurrently.
4. **Backward Reset**
   In a storage-free Telescope GasP module, the backward transferred LO acknowledge signal on the predecessor statewire turns off its own LO drive after 5 gate delays. Traditional GasP modules, like module Store, turn off their forward HI and backward LO drives concurrently.

The two transfer parts each contain a driver and a half-keeper. Similar to the half-keeper design organizations of 4-2 Gasp in [18] and of 6-4 GasP in [7], the half-keeper serves two purposes: it prevents the statewire from floating when the drive ceases, and it avoids a keeper-driver short-current conflict.

One of the design criteria for the current Telescope GasP implementations is a small 2 gate delay latency between incoming and outgoing handshakes. From [8], we know that we need *at least* 2 gate delays to implement non-inverting Telescope GasP. Our design criterion is to make this latency *no more than* 2 gate delays. To be more precise: we want a minimum latency of 2 gate delays per module from the moment its last parallel incoming handshake goes HI (starts) to the moment its outgoing handshakes go HI (start), and, likewise, a minimum latency of 2 gate delays from the moment its last outgoing handshake goes LO (resets) to the moment its parallel incoming handshakes go LO (reset).

To guarantee this low latency, some of the control functionality is inevitably forced into the drivers for the handshake channels. This results in a myriad of custom driver designs, each with 1 gate delay latency. Section 2 below lists all custom design drivers used for the design of RepeatUntil and RepeatWhile and includes their circuit schematics provided these are not already given in [9] or [10].

2. Custom Driver Designs for Repeat Modules

To achieve a small 2 gate delay transfer latency, some of the forward and backward transfer logic is inevitably integrated into the driver and half-keeper logic. As a result, Telescope GasP modules use a large variety of driver and half-keeper circuits for incoming and outgoing statewires - larger than we are used to see in traditional, e.g. 6-4, GasP modules.

Section 2.1 gives the implementations of HI driver and LO half-keeper logic for successor drivers of outgoing statewires of Repeat modules. Section 2.2 does the same for the LO driver and HI half-keeper logic for predecessor drivers of incoming statewires. By convention, the initial value of a statewire is LO, and so the **master clear (mc)** signal and related circuitry for fast return-to-zero resetting appear in the predecessor drivers.

2.1. Single-Track Statewire Drivers for Pull-HI and Keep-LO

Our implementations of RepeatWhile and RepeatUntil use the following HI driver and LO half-keeper logic for successor drivers:

- **Succ AND Driver:** The AND of two LO inputs drives the statewire HI, and a half-keeper keeps it LO. See [9] for circuit details.
- **Succ OR1xAND Driver:** 3 LO inputs ORAND-ed to drive the statewire HI; half-keeper to keep it LO.

Both circuits have an input-to-output transition time of one gate delay. The new driver implementation for Succ OR1xAND Driver follows in
Figure 1 below.

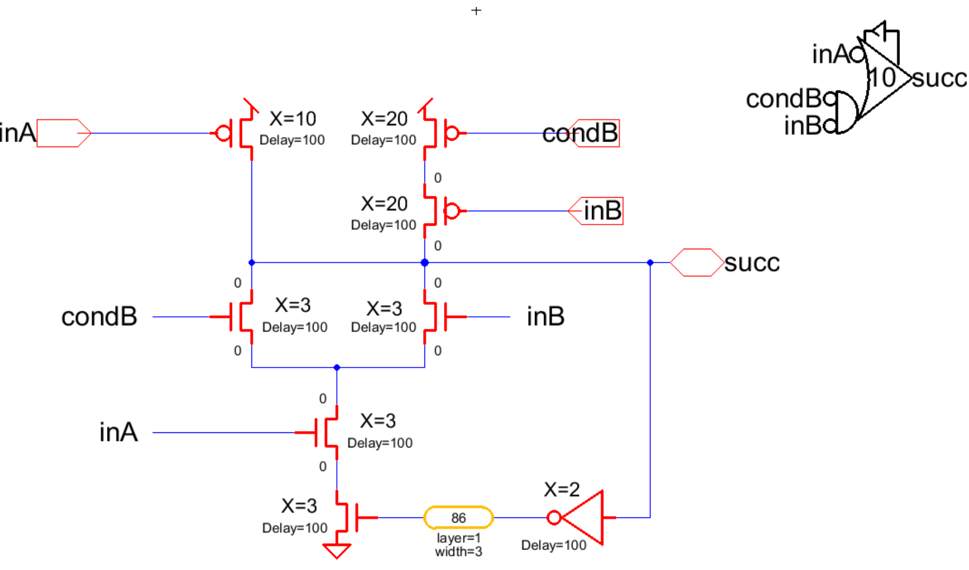


Figure 1: Transistor-level GasP design for **Succ OR1xAND Driver**. Either a LO input on **inA** or LO inputs on both **inB** and **condB** will drive statewire **succ** HI within one gate delay. The half-keeper is there to keep **succ** LO, when needed. We use the icon in the top right corner of the picture as a gate-level shortcut for this driver. It is used in the RepeatUntil module.

2.2. Single-Track Statewire Drivers for Pull-LO and Keep-HI

Our implementations of RepeatUntil and RepeatWhile use the following LO driver and HI half-keeper logic for predecessor drivers:

- **Pred Driver:** A HI input drives the statewire LO, and a half-keeper keeps it HI - see [9] for details.
- **Pred OR Driver with MC and fastRTZ:** Pred OR Driver with extra 2-input guarded fastRTZ facility. Two HI inputs, either of which drives the statewire LO for normal resets, and half-keeper to keep it HI. See [10] for circuit details.

Both circuits have an input-to-output transition time of one gate delay. Their implementations are given in [9,10]. By convention, statewires are initialized LO. So, unless otherwise specified, LO drivers have a **master clear (mc)** input signal and related circuitry to initialize the statewire to LO. The driver implementations for the Repeat modules follow this convention. That means that also any additional circuitry for fast (return-to-zero) resetting goes into the LO driver logic of the predecessor drivers.

3. Repeat Module

Repeat modules are used in ARCwelder [5,6] to implement repetitive data streaming. We implemented two types of Repeat module, only one of which was listed in [5]. We dubbed the listed version RepeatUntil, and added RepeatWhile. Implementation details for the RepeatWhile module follow in Section 3.1 below, and implementation details for the RepeatUntil module follow in Section 3.2 below.

3.1. RepeatWhile

The RepeatWhile module streams its source data zero or more times to its outgoing handshake channel. Initially, and again after each output, it waits for a guard input to determine whether or not to stream the source data at its incoming channel to its outgoing channel one more time.

The implementation of the RepeatWhile module in **Figure 2** follows the second design approach for designing Telescope GasP modules – see [8] for a quick explanation of the two design approaches. Because the second approach gives a simpler solution than the first approach when it comes to implementing a RepeatWhile, this is the only implementation we will provide. The implementation has two self-resetting loops. The top loop has separate taps to start and stop the HI respectively LO drive for *out[sw]* respectively *guard[sw].* The bottom loop is used to start and stop the LO drive for *in[sw].*

SPICE-level simulations for this implementation follow in **Figure 3** and **Figure 4. Figure 3** shows the handshake behavior, while **Figure 4** shows some of the details for the internal signaling behavior.

The handshake behavior can be described as follows. Initially, **in[sw]**, **guard[sw]**, **out[sw]**, **mc** and **fastRTZ** are LO. When **in[sw]** and **guard[sw]** go HI, indicating the presence of both source data and the fact that a guarded decision has been made, and **out[sw]** is still LO, indicating the availability of space, the following actions take place **in sequence** and ordered as indicated:

1. If the guarded decision is a GO, i.e., if **guard[d]** is HI, then the Forward Transfer part streams the source data to the successor module by driving **out[sw]** HI, 2 gate delays later. If the guarded decision is a NOGO, i.e., if **guard[d]** is LO, then the repetitive sequence of actions 2 to 5 to 1 is skipped for the following two final actions:
   - 1.1. Start both Backward Transfer parts by driving **in[sw]** and **guard[sw]** LO, 5 gate delays after the last of them went HI. The additional waiting time serves the purpose of accommodating a 5 gate delay HI pulse on **in[sw]** and **guard[sw]**.
   - 1.2. Then start both Backward Resets in order to limit the LO drives on **in[sw]** and **guard[sw]** to a 5 gate delay pulse.
2. The Forward Reset part kicks in, limiting the HI forward drive to a 5 gate delay pulse signal.
3. The Backward Transfer part waits until **out[sw]** is LO, and then requests a new guarded decision by driving **guard[sw]** LO, 2 gate delays later.
4. The Backward Reset kicks in, limiting the LO drive on **guard[sw]** to a 5 gate delay pulse.
5. The module now waits until guard[sw] is HI again, and then continues with action 1 above.

After action 1.2, at the end of this variable-length sequence of actions, the RepeatWhile module waits until both **in[sw]** and **guard[sw]** are HI again, so it can start its next RepeatWhile cycle.

The Forward Transfer part in **Figure 2** uses a Succ AND Driver [9]. There is 1 additional gate between incoming handshake channels **in[sw]** and **guard[sw]** and the input of the Succ AND Driver, giving a total forward latency of 2 gate delays. The Backward Transfer part for **guard[sw]** uses a Pred OR Driver with master clear and fast reset [10]. There is 1 additional gate between outgoing handshake channel **out[sw]** of the RepeatWhile module and the input of the Pred OR Driver with fast reset, giving a total backward latency of 2 gate delays. Incoming handshake channel **in[sw]** has no dependencies to out[sw]. It plays a role in ending the repetition, as explained in items 1.1 and 1.2 above, and adds no extra backward latency to the 5 gate delay HI handshake pulses for **in[sw]** and **out[sw].**

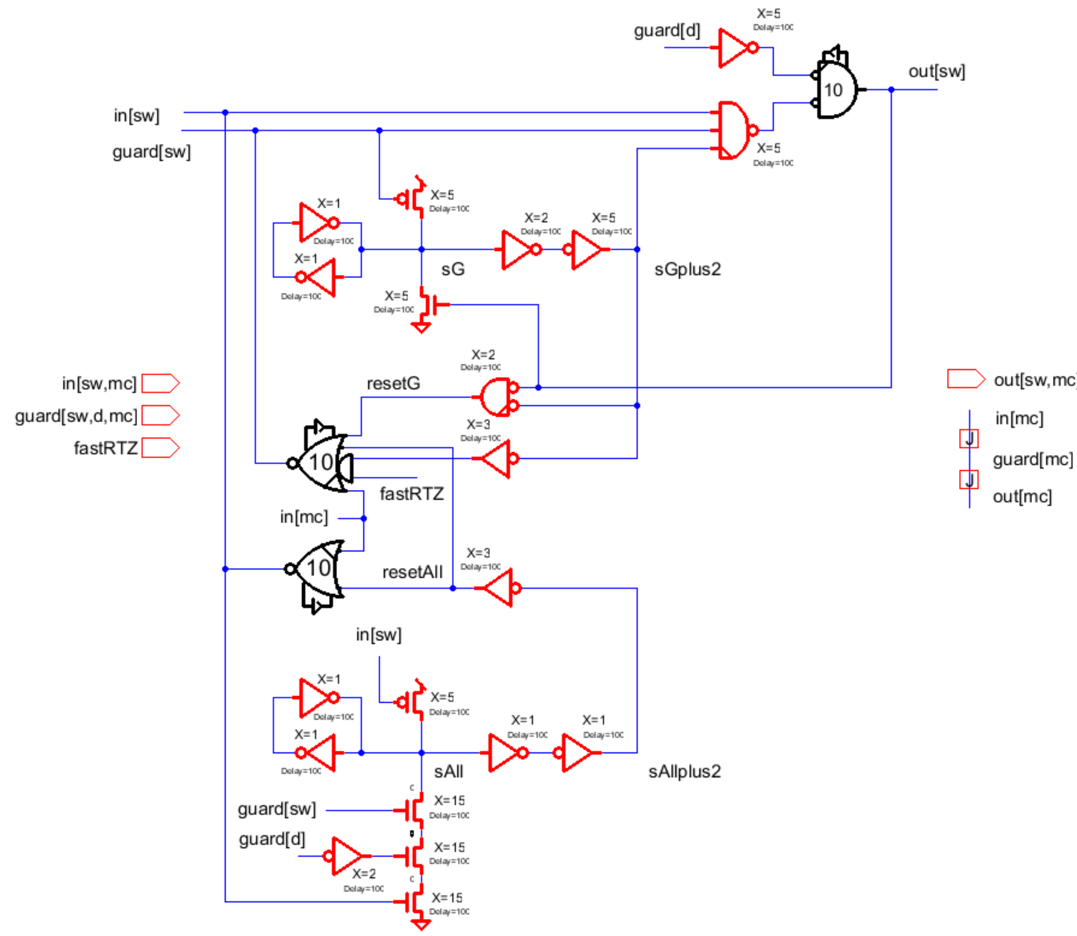


Figure 2: Telescope GasP implementation for the RepeatWhile module. It has two input statewires with a master clear signal, **in[sw,mc]** and **guard[sw,mc]**, and one output statewire with a joined master clear signal, **out [sw,mc]**, and a fast reset input **fastRTZ** The data signal in the guard **guard[d]** is bundled with the statewire signal in the guard **guard[sw]**. The implementation is saved under TelescopeGasP_wStateSharing, under module name RepeatWhileTfastRTZ. Substring `TfastRTZ' in the module name indicates that this is a Telescope GasP implementation with fast reset capability. The module has an internal reset signal, used during normal telescope handshakes involving **out[sw]**, and an external reset signal for fast resetting. To ensure that we reset only actions related to the current iteration cycle, the fast reset and internal reset signals are connected *ONLY* to the predecessor driver for **guard[sw]** and are AND-ed with an internal guard signal tapped from internal statewire **sG**.

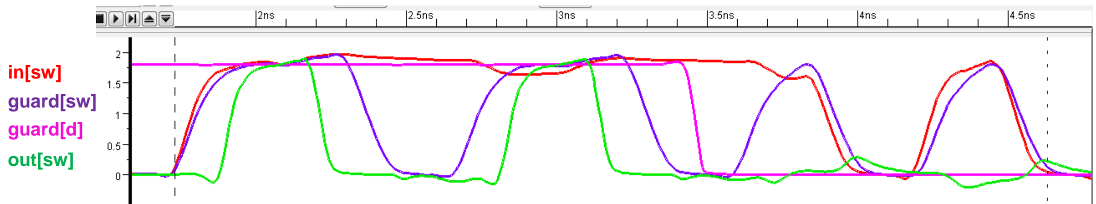


Figure 3: SPICE-level simulations of the RepeatWhile module in **Figure 2**. If both in[sw] and guard[sw] are HI then out[sw] will go HI provided pink signal guard[d] is HI, and if guard[d] is LO then out[sw] will stay LO and both in[sw] and guard[sw] will go LO. Guard condition, guard[d], may change when guard[sw] is LO. For every HI pulse on out[sw], the source data provided via in[sw] (not shown here) are streamed to the destination of out[sw]. For every HI pulse on out[sw] there is an encompassing HI pulse on guard[sw]. The simulation shows:

- One RepeatWhile loop with two repetitive out[sw]-guard[sw] handshake telescopes during which guard[d] is HI and which ends with guard[d] going LO followed by a final guard[sw] HI pulse and in[sw] going LO.
- A subsequent RepeatWhile loop during which guard[d] is LO, which ends immediately with a HI pulse on both in[sw] and guard[sw], while out[sw] remains LO.

Note that (1) each HI pulse on out[sw] is strictly contained within a HI pulse on in[sw] and guard[sw], and that (2) out[sw] participates in the telescope action if and only if in[sw], guard[sw] and guard[d] are all HI.

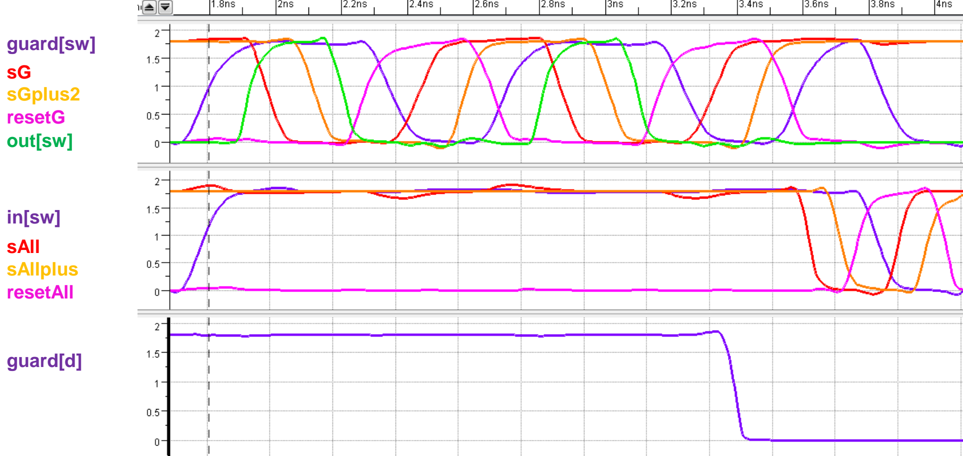


Figure 4: SPICE-level simulations showing the internal signaling details for module RepeatWhile in **Figure 2** for the first of the two RepeatWhile loops in **Figure 3**.

- The bottom window shows guard[d] changing from HI to LO after the second HI pulse on guard[sw].
- The top window shows the signaling details for guard[sw] to out[sw] handshakes. The top simulation starts with the purple guard[sw] going HI, followed by the green out[sw] going HI 2 gate delays later, followed by the internal sG signal in red going LO 1 more gate delay later and the internal sGplus2 signal in orange going LO 2 more gate delays later, and back to the green out[sw] going LO, 5 gate delays after it went HI, and to the purple guard[sw] signal going LO, also 5+4=9 gate delays after it went HI. The safety margin of 3 gate delays for resetting guard[sw] that we see in **Figure 2** when going from out[sw] HI disabling the inverted input AND gate to sGplus2 LO enabling the AND is large enough to guarantee a telescopic handshake relation between guard[sw] and out[sw]. It is also small enough to fit within the minimum HI pulse width of 5 gate delays for out[sw] in order to maintain throughput. The remaining 2 gate delays after sGplus2 goes LO are used to stop the HI drive on out[sw] so the receiver can reset out[sw] without a fight conflict. When out[sw] is reset LO, the inverted input AND gate makes resetG go HI one gate delay later, which then resets guard[sw] to LO one more gate delay later - two gate delays after out[sw] went LO. After guard[d] changes to LO, guard[sw] handshakes are reset LO without incurring a handshake on out[sw].
- The middle window shows the signaling details for in[sw] handshakes. The simulation starts with the purple in[sw] going HI, and is followed by the red sAll signal going LO after guard[d] has fallen and guard[sw] has become HI again, as indicated in the top and bottom windows. Signal sAll going LO makes orange signal sAllplus2 go LO 2 gate delays later, which makes pink signal resetAll go LO 1 more gate delay later, which gets us back to the purple signal in[sw] that goes LO 1 more gate delay later, at about the same time that the guard[sw] goes LO in the top window.

Without a fast reset signal, the minimum cycle time is min(5, k*(2+5+2+5))=min(5,k*14) gate delays, where k is the number of **out[sw]** iterations. The cycle time is set by the 2 gate delay transfers and the 5 gate delay loops in the circuit diagram of **Figure 2,** assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of k*4 gate delays for each successive non-Repeat Telescope GasP module between the current RepeatWhile and the next Store module. In case of nested Repeat modules, a recursive formula can be written to calculate the overall cycle time.

Using a fast reset signal, we can reduce the backward latency and reduce the cycle time to min(5,k*12) with an increase in steps of k*2 gate delays for each successive non-Repeat Telescope GasP module between the current RepeatWhile and the next Store module. We designed a test environment that generates such a fast reset signal and passes it to the RepeatWhile module via the already provided fastRTZ input. We tested the behavior of the RepeatWhile module with this fast reset, using the configuration in **Figure 5**.

Figure 5 shows a set of parallel sender queues on the top, designed using traditional 6-4 GasP. The queues feed the two pairs of source input and guard input channels of the nested RepeatWhile data flow queue in the middle. The bottom portion of **Figure 5** contains the receiving half of a Store module [9]. We deleted the sending half from the Store implementation, because we don't need it, and we added extra circuitry to generate a fast reset signal, which is called **fastRTZ**.

This test configuration works as follows. Four gate delays after the start of an incoming handshake to the Store module, i.e., after **out[sw]** goes HI, the Store module makes **fastRTZ** go HI. The **fastRTZ** signal is distributed to the rightmost RepeatWhile module in the inner loop of the green data path under test and to its successor Telescope GasP modules - i.e., to the rightmost Merge module. At the same time, i.e., four gate delays after **out[sw]** goes HI, the Store module makes its local clock signal **cl** HI. The **cl** signal acts as input to the LO driver for **out[sw].** Consequently, five gate delays after **out[sw]** goes HI, all active handshake communications in the rightmost RepeatWhile and Merge modules are reset concurrently. The reset stops after 5 gate delays, via the self-resetting loop in the Store module.

Figure 6 shows SPICE-level simulation waveforms for **fastRTZ** and the corresponding concurrent reset to LO for channels **mid3a1[sw]** and **guard1[sw]** going out of respectively into the rightmost RepeatWhile module. Note that the leftmost RepeatWhile and Merge modules are not affected by this **fastRTZ.** The two fast reset inputs to the leftmost Merge modules are tied to ground in **Figure 5**. If needed, we could generate a fast reset signal for the leftmost RepeatWhile and Merge modules from the internal reset signal **resetAll** in the rightmost RepeatWhile module that resets **mid2b[sw]** as well as **guard1[sw]**.

RepeatWhileTfastRTZ_test_fastRTZ

smg-mr 23 Oct 2011

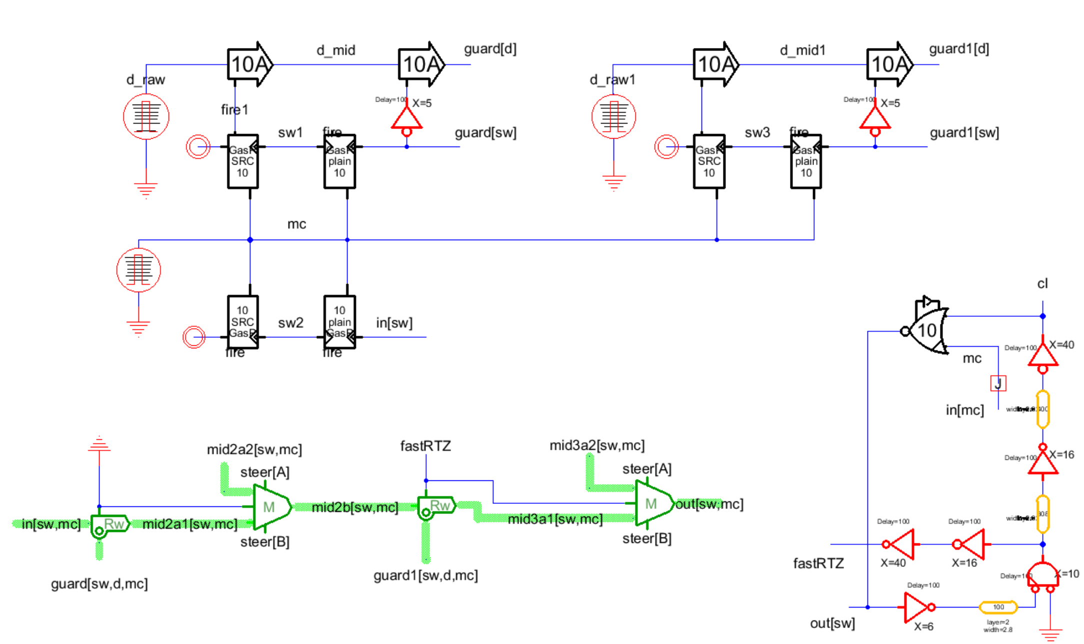


Figure 5: Fast reset configuration and test environment for the bottom-left green data path with RepeatWhile and Merge modules. The green icons with text `M' represent the Merge modules. Those with text `Rw' represent the RepeatWhile modules. The above test configuration concurrently resets statewires guard1[sw], mid3a1[sw] and out[sw] that are connected to the rightmost RepeatWhile and Merge modules that form the inner loop in the green data path under test. Note that, unlike the fast reset signal **mc_fastRTZ** in [9], but similar to the fast reset signal **fastRTZ** in [10], the **fastRTZ** signal generated by the above Store module is independent of master clear signal **mc**. Note also that **fastRTZ** is distributed only to the rightmost RepeatWhile and Merge. The fast reset inputs for the leftmost RepeatWhile and Merge are grounded. This guarantees that only inner loop actions are reset by this **fastRTZ** signal, while outer loop actions remain unaffected by it.

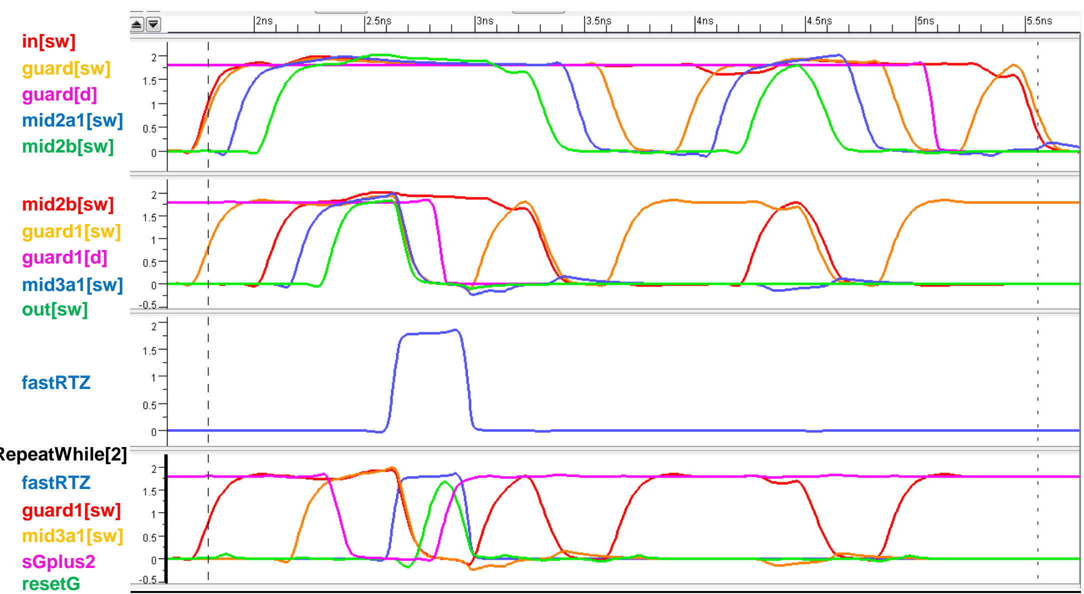


Figure 6: SPICE-level simulation results for the fast reset and test configuration in **Figure 5**.

- The third window from the top shows the fastRTZ pulse.
- The top window shows that fastRTZ does not impact the leftmost RepeatWhile and Merge modules in the outer loop of the data path under test. This is exactly how we expect it to be, because the fast reset inputs for these components are connected to ground. The backward transfer delay from handshake signal mid2b[sw], in green and going out of the first Merge module, to handshake signal mid2a1[sw], in blue and coming into the first Merge, remains 2 gate delays. The backward transfer delay from the blue handshake signal mid2a1[sw] going out of the first RepeatWhile module to the orange handshake signal guard[sw] coming into the module also remains 2 gate delays.
- The second window from the top shows that the fastRTZ *does* impact the rightmost RepeatWhile and Merge modules in the inner loop of the data path under test. When fastRTZ goes HI, it concurrently resets all inner loop handshakes signals between guard1[sw] and out[sw] that caused out[sw] to rise. Specifically, it resets out[sw] in green, mid3a1[sw] in blue, and guard[sw] in orange. Note that all three signals are LO by the end of the fastRTZ HI pulse.
- The bottom window shows the internal reset behavior for the rightmost, RepeatWhile module in **Figure 5**.
  - Blue signal fastRTZ is the fast reset signal generated by the Store module. During its 5 gate delay HI pulse, the orange mid3a1[sw] and red guard1[sw] handshake signals are reset LO, simultaneously.
  - Green signal resetG is part of the normal backward reset path. When HI, it resets the red incoming handshake signal of the RepeatWhile module to LO, with or without the extra help of the guarded fastRTZ signal. From the RepeatWhile schematics in **Figure 2**, we can see that the guards for the internal reset and the fast reset originate from the same signal: sGplus2. If the guard holds, i.e., if sGplus2 is LO, then the internal reset resetG rises 1 gate delay after a falling transition on the orange outgoing handshake, and 2 gate delays after the blue fast reset rises.
  - As in [10], we use a strong fast reset signal that functions according to the logical effort expectations of a normal post-initialized Telescope GasP operation. Consequently, the red incoming handshake signal for guard1[sw] has a steep reset slope when reset by fastRTZ. As a result, the red signal is largely reset by the time the green internal reset starts, making the internal reset less effective during fast resets of RepeatWhile.
  - Pink signal sGplus2 acts as guard for both the fast and the internal reset. If it is LO at the start of the fastRTZ HI pulse, then the red guard1[sw] signal will be reset. In that case, it goes HI 4 gate delays after fastRTZ goes HI, and 1 more gate delay later it stops the HI pulse on the green internal reset signal and its stops the fastRTZ drive to guard1[sw], thus ceasing the LO drive for guard1[sw]. The 5 gate delay HI pulse on fastRTZ stops around the same time.

3.2. RepeatUntil

The RepeatUntil module streams its source data one or more times to its outgoing handshake channel. After each output, it waits for a guard input to determine whether or not to stream the source data at its incoming channel to its outgoing channel one more time.

The implementation of the RepeatUntil module in **Figure 7** below follows the second design approach for designing Telescope GasP modules – see [8] for a quick explanation of the two design approaches. The implementation resembles that of the Toggle Merge design in **Figure 26** of [10]. It has two self-resetting loops for external statewires, with separate taps to start and stop the HI drive for **out[sw]** and the LO drives for **in[sw]** and **guard[sw]**. It has two additional loops to encode internal statewires, called **RR1** and **RRmore** that keep track of whether the source data are forwarded for the first time or for more times.

The handshake behavior can be described as follows. Initially, **in[sw]**, **guard[sw]**, **out[sw]**, **mc** and **fastRTZ** are LO, **RR1** is high, and **RRmore** is LO. When **in[sw]** goes HI, indicating the presence of source data, and **out[sw]** is still LO, indicating the availability of space, the following actions take place **in sequence** and ordered as indicated:

1. The Forward Transfer part streams the source data to the successor module by driving **out[sw]** HI, 2 gate delays after **in[sw]** goes HI.
2. The Forward Reset part kicks in, limiting the HI forward drive to a 5 gate delay pulse signal. The Forward Reset works via internal statewire **RR1,** which is reset to LO 1 gate delay after **out[sw]** goes HI, and which in turn resets **RR1plus2** to LO 2 gate delay later, which results in stopping the LO drive for **out[sw]** 2 more gate delays later – 5 gate delays after **out[sw]** went HI.
3. The module waits until it can do a Backward Transfer. The Backward Transfer works via statewire **RRmore,** which is set to HI 1 gate delay after **out[sw]** goes LO, and which in turn sets **RRmoreplus2** to HI 2 gate delay later – 3 gate delays after **out[sw]** went LO. With **RRmoreplus2** HI, the module is ready to follow a guarded decision on doing another **out[sw]** handshake cycle.
4. The module waits until **guard[sw]** is HI. If the guarded decision is a GO, i.e., if **guard[d]** is HI, then the module proceeds with the repetitive sequence of actions 5 to 9 to 4. If it's a NOGO, i.e., if **guard[d]** is LO, the module skips the repetition and ends with the following final series of actions:
   - 4.1. First start both Backward Transfer parts by driving **in[sw]** and **guard[sw]** LO, 5 gate delays after the last of them went HI. The 5 gate delays serves the purpose of accommodating a 5 gate delay HI pulse on **in[sw]** and **guard[sw**], and are set by the top-right loop in **Figure 7** via internal reset signal **resetAll** which feeds both predecessor drivers.
   - 4.2. Then start both Backward Resets in order to cut off the LO drives on **in[sw]** and **guard[sw]** to a 5 gate delay pulse. This is again accomplished by the top-right loop in **Figure 7**, which resets the HI **resetAll** signal to LO after 5 gate delays and thus cuts off the LO drives for **in[sw]** and **guard[sw]** one gate delay later – 5 gate delays after both went LO.
   - 4.3. Meanwhile, 3 gate delays after **resetAll** goes HI and hence 2 gate delays after **in[sw]** and **guard[sw]** go LO, **RRmoreplus2** goes LO and puts on hold successive **guard[sw]**-**out[sw]** telescope handshakes. One more gate delay later, i.e., 3 gate delays after **in[sw]** and **guard[sw]** go LO, **RR1plus2** goes HI. The module is now ready to respond to the next **in[sw]** handshake, with 5-3=2 gate delays to spare before **in[sw]** can be expected to go HI again.
5. We have a guarded decision that's a GO, **in[sw]** is HI, **out[sw]** is LO for at least 3 gate delays, and **RRmoreplus2** is HI. The module is poised for another **out[sw]** handshake, which will start two gate delays after **guard[sw]** rises, thus allowing **out[sw]** to be LO for at least 5 gate delays.
6. The Forward Transfer in item 4 above is followed by a Forward Reset through the bottom-right loop in **Figure 7**, which cuts off the HI drive for **out[sw]** after 5 gate delays.
7. The Backward Transfer part waits until **out[sw]** is LO, and then requests a new guarded decision by driving **guard[sw]** LO, 2 gate delays later.
8. The Backward Transfer in item 6 above is followed by a Backward Reset through the bottom-right loop in **Figure 7**, which cuts off the LO drive for **guard[sw]** after 5 gate delays.
9. The module continues with action 4 above.

After action 4.3, at the end of this variable-length sequence of actions, the RepeatUntil module waits until both **in[sw]** is HI again, so it can start its next RepeatUntil cycle.

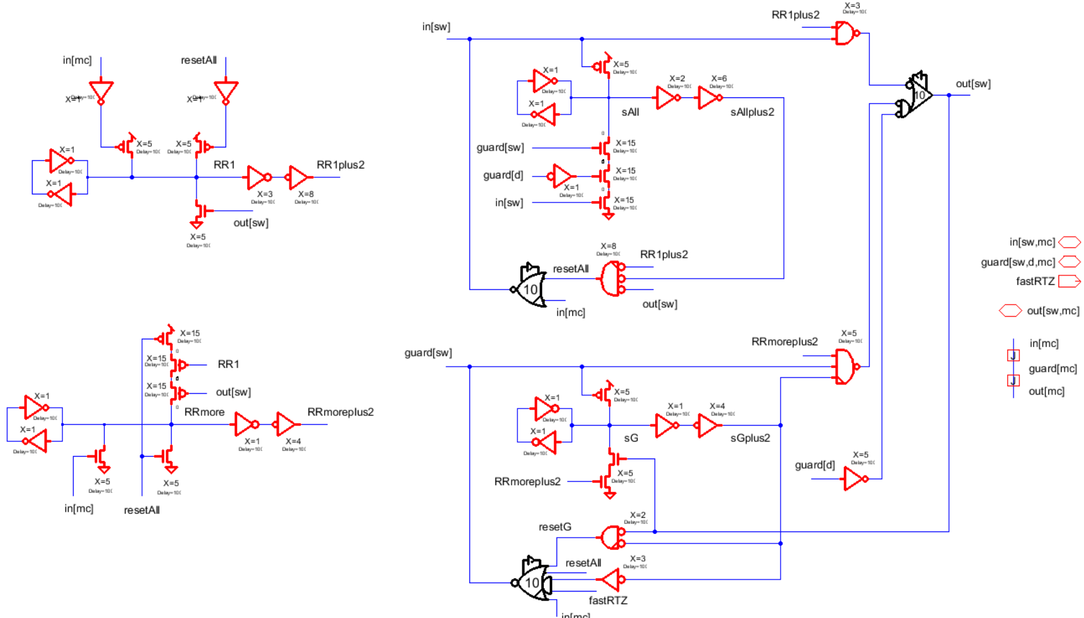


Figure 7: Telescope GasP implementation for the RepeatUntil module. It has two input statewires with a master clear signal, **in[sw,mc]** and **guard[sw,mc]**, one output statewires with a joined master clear signal, **out [sw,mc]**, and a fast reset input **fastRTZ** The data signal in the guard **guard[d]** is bundled with the statewire signal in the guard **guard[sw]**. The implementation is saved under TelescopeGasP_wStateSharing, under module name RepeatUntilTfastRTZ. Substring `TfastRTZ' in the module indicates that this is a Telescope GasP implementation with fast reset capability. Similar to the RepeatWhile implementation in **Figure 2**, there is an internal reset signal, used during normal telescope handshakes with **out[sw]**, and an external reset signal for fast resetting. To ensure that we reset only actions related to the current iteration, the fast reset and internal telescope reset signals are connected *ONLY* to the predecessor driver for **guard[sw]** and are AND-ed with an internal guard signal tapped from signal **sG**.

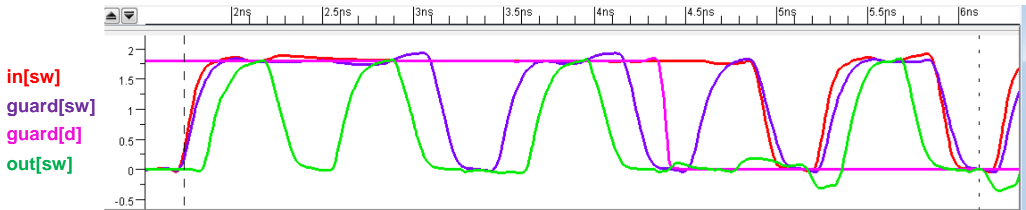


Figure 8: SPICE-level simulations of the RepeatUntil module in **Figure 7**. Each in[sw] pulse in red contains at least one out[sw] HI pulse in green. Another green out[sw] HI pulse is added for every guard[sw] pulse in purple with pink guard[d] HI, until in[sw] is HI and guard[sw] is HI but guard[d] is LO. Guard condition, guard[d], may change when guard[sw] is LO. For every HI pulse on out[sw], the source data provided via in[sw] (not shown) are streamed to the destination of out[sw]. The simulation shows:

- One RepeatUntil loop consisting of one in[sw] handshake inside which there is a single out[sw] handshake followed by two guard[sw]-out[sw] handshake telescopes during which guard[d] is HI, followed by guard[d] going LO, followed by a final guard[sw] HI pulse and in[sw] going LO.
- A subsequent RepeatWhile loop during which guard[d] is LO, consisting of one in[sw] handshake inside which there is a single out[sw] handshake and separate of which there is a guard[sw] handshake that ends simultaneously with the handshake on in[sw].

Unlike suggested by these waveforms, a HI pulse on out[sw] must be strictly contained within a HI pulse on guard[sw] only from the second to the last out[sw] HI pulse in an in[sw] HI pulse.

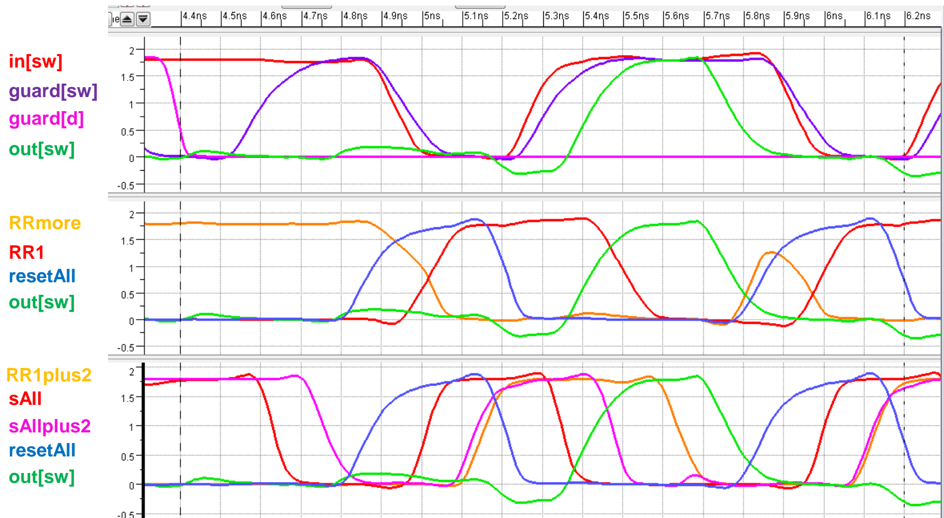


Figure 9: SPICE-level simulations showing the internal signaling details for module RepeatUntil in **Figure 7** between the end of the first RepeatUntil loop and the start of the next RepeatUntil loop in **Figure 8**.

- The top window shows the external handshake signals, just like in **Figure 8**.
- The middle window shows the signaling details for the internal statewires RRmore and RR1. When resetAll goes HI at the end of first RepeatUntil loop, RRmore goes LO 1 gate delay later and RR1 goes HI 2 gate delays after resetAll goes HI. Both RRmore and RR1 are state holding. So, by the end of the 5 gate delay HI pulse on resetAll, RR1 and resetAll remain LO respectively HI. Internal statewire RR1 is reset to LO as soon as out[sw] goes HI in the next RepeatWhile loop. When out[sw] goes LO, it sets RRmore to HI and resetAll to LO, 1 gate delay later. With resetAll LO, RRmore is reset back to LO after 1 more gate delay. This explains the small rising pulse on RRmore, which his harmless. RRmore is LO and RR1 high by the end of the resetAll pulse and well before the next in[sw] and guard[sw] handshakes come in.
- The bottom window shows the signaling details related to RR1 and sAll. When resetAll goes HI at the end of the first RepeatUntil loop, sAllplus2 and RR1plus2 go HI 4 gate delays later, which is 1 gate delay before the end of the resetAll HI pulse and 2 gate delays before the next in[sw] handshake starts. Signal sAll goes LO as soon as it becomes clear that the current out[sw] pulses will not be followed by more out[sw] pulses in the current RepeatWhile loop, i.e., when in[sw] and guard[sw] are HI and guard[d] is LO. Here, sAll and sAllplus2 go LO right at the start of the loop's first out[sw] HI pulse. Signal RR1plus2 is part of the 5 gate delay self-resetting loop for out[sw]: RR1plus2 goes LO 3 gate delays after out[sw] goes HI and stops the out[sw] HI drive 2 gate delays later.

SPICE simulations for the module implementation in **Figure 7** are presented in **Figure 8** and **Figure 9**. **Figure 8** shows the handshake behavior, and **Figure 9** shows details about the signaling behavior internal to the module.

Regarding forward and backward transfer latencies, we can say the following. The Forward Transfer part in **Figure 7** uses the Succ OR1xAND Driver of

Figure 1 in this document. There is 1 additional gate between incoming handshake channels **in[sw]** and **guard[sw]** and the input of the Succ AND Driver, giving a total forward latency of 2 gate delays. The Backward Transfer part for **guard[sw]** uses a Pred OR Driver with master clear and fast reset [10]. There is 1 additional gate between outgoing handshake channel **out[sw]** of the module and the input of the Pred OR Driver with fast reset, giving a total backward latency of 2 gate delays. Incoming handshake channel **in[sw]** plays a role in setting up the guarded repetitions and ending the overall repetition, as explained in items 4.1 to 4.3 above, and adds no extra backward latency to the 5 gate delay HI handshake pulses for **in[sw]** and **out[sw]**.

Without a fast reset signal, the minimum cycle time is min(5, k*(2+5+2+5))=min(5,k*14) gate delays, where k is the number of **out[sw]** iterations. The cycle time is set by the 2 gate delay transfers and the 5 gate delay loops in the circuit diagram of **Figure 7,** assuming that successor and predecessor modules keep the statewires HI respectively LO for a minimum of 5 gate delays. The cycle time increases in steps of k*4 gate delays for each successive non-Repeat Telescope GasP module between the current RepeatUntil module and the next Store module. In case of nested Repeat modules, a recursive formula can be written to calculate the overall cycle time.

Using a fast reset signal, we can reduce the backward latency and reduce the cycle time to min(5,k*12) with an increase in steps of k*2 gate delays for each successive non-Repeat Telescope GasP module between the current RepeatUntil and the next Store module. We designed a test environment that generates such a fast reset signal and passes it to the RepeatUntil module via the already provided fastRTZ input. We tested the module under this fast reset, using the configuration in **Figure 10**, which is very similar to the configuration in **Figure 5** for testing RepeatWhile module under a fast reset.

Figure 11 and **Figure 12** show the SPICE-simulated waveforms. Note that only the rightmost RepeatUntil and Merge modules in the innermost loop of the green data path are affected by the **fastRTZ** signal. We tied the two fast reset inputs for the leftmost RepeatUntil and Merge modules in the outermost loop to ground. If needed, we could generate a fast reset signal for the leftmost RepeatUntil and Merge modules from the internal reset signal **resetAll** in the rightmost RepeatUntil module.

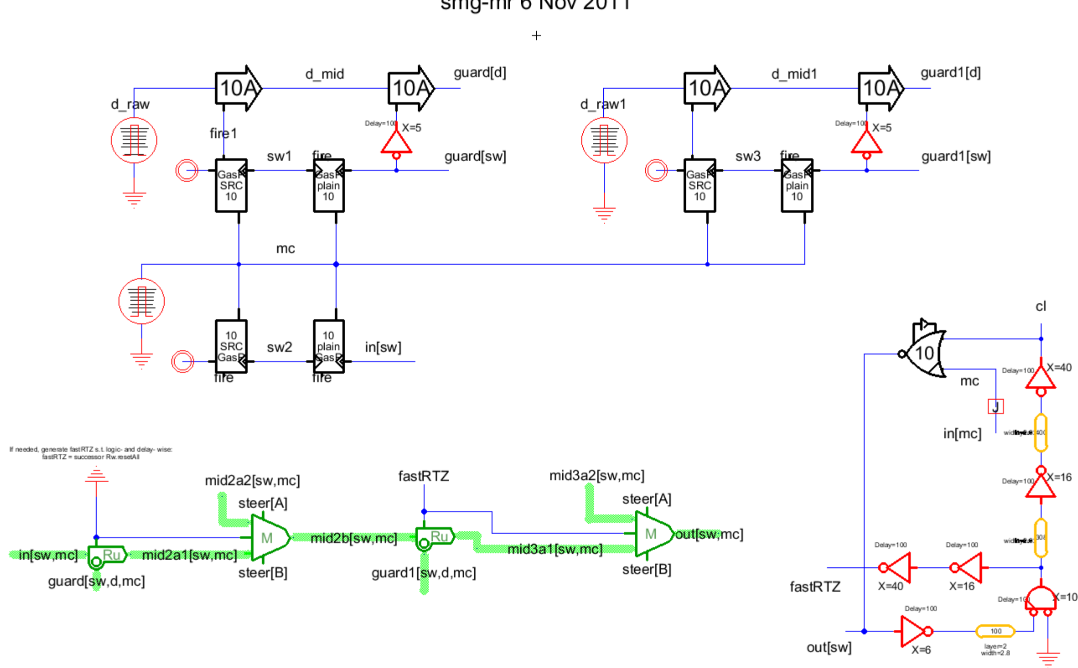


Figure 10: Fast reset configuration and test environment for the bottom-left green data path with RepeatUntil and Merge modules. The green icons with text `M' represent the Merge modules. Those with text `Ru' represent the RepeatUntil modules. The above test configuration concurrently resets statewires guard1[sw], mid3a1[sw] and out[sw] that are connected to the rightmost RepeatWhile and Merge modules that form the inner loop in the green data path under test. This reset and test configuration is similar to the one in **Figure 5** with RepeatWhile modules. The **fastRTZ** signal generated by the above Store module is independent of master clear signal **mc**, and distributed only to the rightmost RepeatWhile and Merge modules. The fast reset inputs for the leftmost RepeatWhile and Merge modules are grounded. This guarantees that only inner loop actions are reset by this fastRTZ signal, while outer loop actions remain unaffected by it.

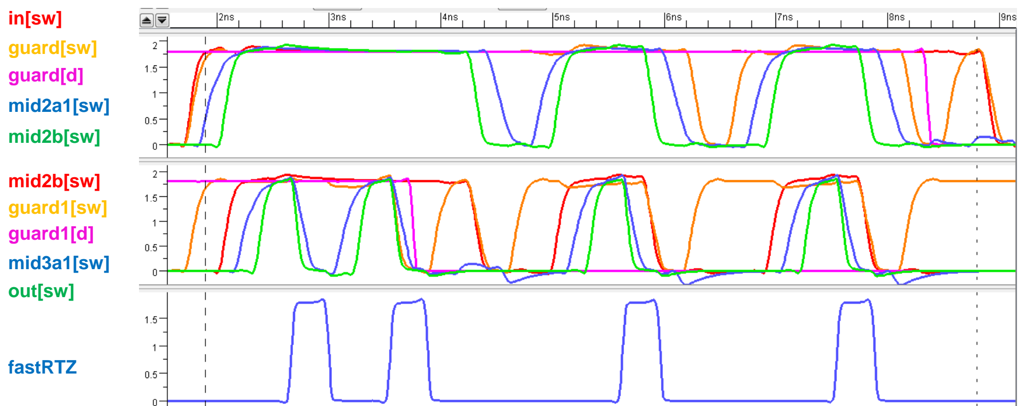


Figure 11: SPICE-level simulation results for the fast reset and test configuration in **Figure 10**.

- The bottom window shows the fastRTZ pulse.
- The top window shows that fastRTZ does not impact the leftmost RepeatWhile and Merge modules that form the outer loop in the green data path under test. This is exactly how we expect it to be given that the fast reset inputs for these components are connected to ground. The backward transfer delay from handshake signal mid2b[sw], in green and going out of the first Merge module, to handshake signal mid2a1[sw], in blue and coming into the first Merge, remains 2 gate delays. The backward transfer delay from the blue handshake signal mid2a1[sw] going out of the first RepeatWhile module to the orange handshake signal guard[sw] coming into the module also remains 2 gate delays.
- The middle window shows that fastRTZ *does* impact the rightmost RepeatWhile and Merge modules that form the inner loop in the green data path under test, as expected. When signal fastRTZ goes HI, it concurrently resets all inner loop handshakes signals between guard1[sw] and out[sw] that caused out[sw] to rise. Specifically, the first fastRTZ HI pulse resets concurrently only the handshakes on the green out[sw] and blue mid3a1[sw] statewires. The second fastRTZ HI pulse resets concurrently out[sw] and mid3a1[sw] as well as the orange guard1[sw] statewire. The third and fourth fastRTZ HI pulses operate while guard1[d] is LO. Consequently, guard1[sw] no longer causes out[sw] to rise, and the third and fourth fastRTZ HI pulses reset concurrently only out[sw] and mid3a1[sw]; guard1[sw] and in[sw] are reset 2 gate delays later via the internal backward reset path in the RepeatWhile module.

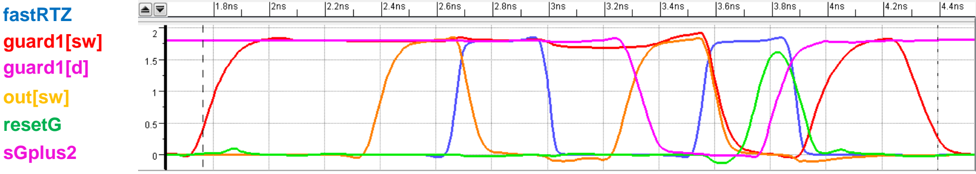


Figure 12: Internal reset behavior for the rightmost RepeatUntil module in the inner loop of **Figure 10**.

- Blue signal fastRTZ is the fast reset signal. During its 5 gate delay HI pulse, the red guard1[sw] signal coming in to the module is reset simultaneously with the orange out[sw] signal going out, provided guard1[sw] played a role in making out[sw] go HI. The first fastRTZ pulse does not reset guard1[sw], but the second fastRTZ pulse does.
- Green signal resetG is part of the normal backward reset path. When HI, it resets the red guard1[sw] to LO, with or without the extra help of the guarded fastRTZ signal. From the RepeatUntil schematics in **Figure 7**, we can see that the guards for the internal reset and the fast reset originate from the same signal: sG, or rather sGplus2. If the guard holds, i.e., if sGplus2 is LO, then resetG rises 1 gate delay after a falling transition on the orange out[sw], and 2 gate delays after the blue fast reset rises.
- As in [10], we use a strong fast reset signal that functions according to the logical effort expectations of a normal post-initialized Telescope GasP operation. Consequently, the red incoming handshake signal guard1[sw] has a steep reset slope, when reset by fastRTZ. As a result, the red signal is largely reset by the time the green internal reset starts, making the internal reset less effective during fast resets of RepeatUntil. We see this happening for the second fastRTZ pulse.
- Pink signal sGplus2 acts as guard for both the fast and the internal reset. If it is LO at the start of the fastRTZ HI pulse, then the red guard1[sw] signal will be reset. In that case, it goes HI 4 gate delays after fastRTZ goes HI, and 1 more gate delay later it stops the HI pulse on the green internal reset signal and its stops the fastRTZ drive to guard1[sw], thus ceasing the LO drive for guard1[sw]. The 5 gate delay HI pulse on fastRTZ stops around the same time.

4. Conclusion and Future Work

This document gives the implementation and simulation details of Telescope GasP modules for RepeatWhile and RepeatUntil for use in ARCwelder [5,6].

For all previous Telescope GasP implementations, we have a minimum cumulative cycle time of 10+4k gate delays, where k is the number of consecutive Telescope GasP modules in the data path following and including the present module up the Store modules that store the computed results at the end of the data path. Repeat modules have a similar formula for cycle time, provided we don't nest them. In case of nested Repeat modules, a recursive formula can be written to calculate the overall cycle time. The increase in cycle time of 4 gate delays per data path module is due to the use of single-track channels.

This puts single-track designs styles, like Telescope GasP, at a disadvantage over other telescoping handshake design styles that use multi-track handshake signaling, such as Click [11]. We can alleviate this disadvantage and obtain a cumulative cycle time of 10+2k gate delays by adding a fast reset strategy.

The fast reset solutions presented in this document share the following key features with [10]:

1. As in [10], the modules in this document use a separate fast reset input that function according to the logical effort expectations of a normal post-initialized Telescope GasP operation.
2. The Repeat modules in this document select which incoming and outgoing handshake channels will participate in the current telescope action. To ensure that only the incoming handshakes that participate and that relate to the current iteration are reset, we AND the fast reset signal with an internal guard signal. The guard indicates whether or not to reset the handshake channel. This guarded reset approach is similar to what we did in [10].

The new fast reset features, unique to Repeat modules are as follows:

1. If there is only one Repeat module, the fast reset signal is generated from Store modules in the data path. As in [9,10], we anticipate that we will be able to identify a particular set of Store modules from which we can tap a fast reset signal, and leave such identification for future work. We distribute this fast reset signal to the Repeat module and all its successor Telescope GasP modules up to the Store module. We can connect the fast rest inputs of components preceding the Repeat module to ground. Alternatively, we could connect these fast rest inputs to the internal reset signal in the Repeat module to support fast resets for pre-repeat actions in the data path.
2. In case of nested Repeat modules, we use the fast reset generation approach described in item 1 above to generate a fast reset signal for the last, i.e., innermost, Repeat module in the data path. We can generate a separate fast reset signal for one of the other nested Repeat modules from the internal reset signal in the next Repeat module that succeeds it, and then distribute this reset signal as fast reset input for all Telescope GasP modules in between the two Repeat modules.
3. We envision that the approach described in items 1 and 2 above would enable a multi-phase fast reset strategy for inner to outer Repeat loops. We leave this vision for future work.

Appendix D

Naturalized Communication and Testing

©2015 IEEE. Reprinted with permission, from the following publication source:

- Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland, “Naturalized Communication and Testing.” *Proceedings of the IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 77–84, 2015.

This IEEE publication is also listed as reference [46] in this thesis.

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.

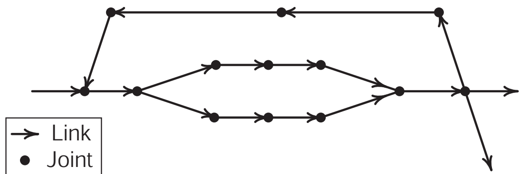


Figure 1 A self-timed dataflow system with communication channels, called links, and flow control and data computation modules, called joints, can be viewed as a directed graph with data flowing in the direction indicated by the arrows.

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

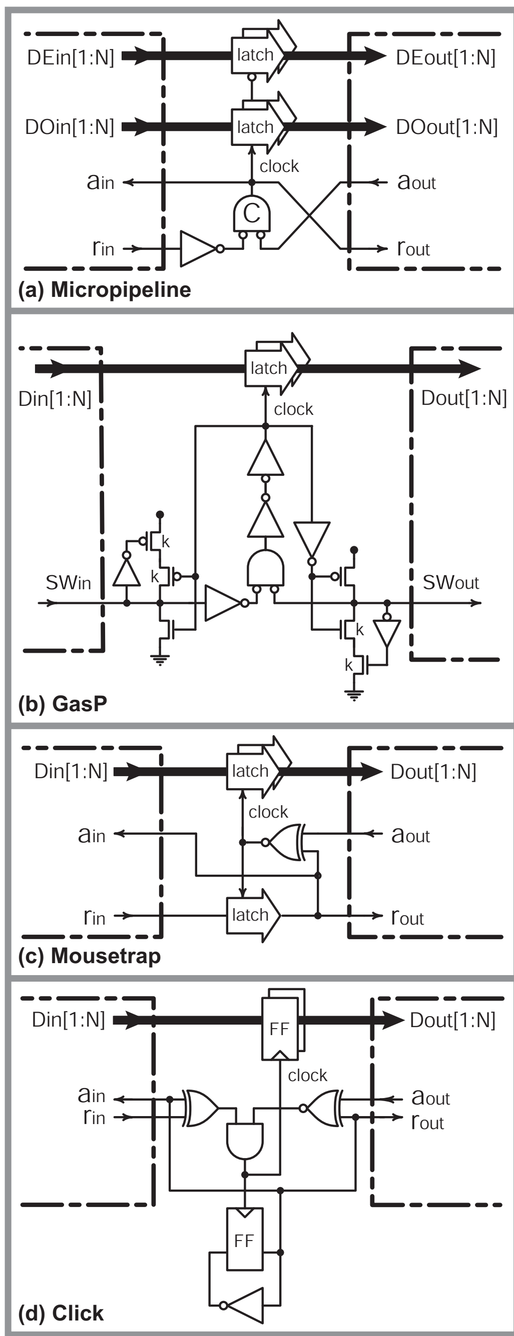


Figure 2 Circuit designs for a single-input single-output FIFO module in each of the four bundled-data two-phase circuit families. The dashed lines mark the interface between the joint in the middle and the left-hand and right-hand links. Note that the joints hold all the logic for flow control and data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

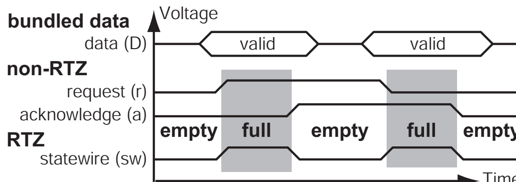


Figure 3 Non-RTZ and RTZ two-phase handshake variants. By convention, data must be valid when the link is full, and may change only when the link is empty. In reality, data may be kited, as long as the values are valid when captured.

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 ($fill_{out}$).

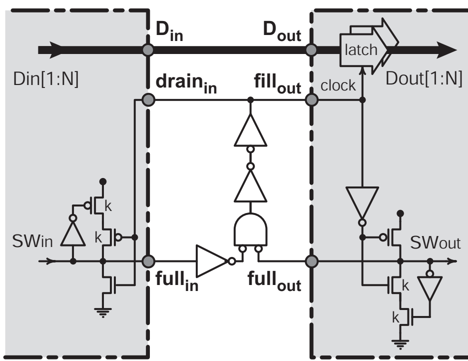


Figure 4 Circuit re-design for the GasP FIFO module of Figure 2(b). The responsibility for capturing data has moved from the joint to the receiving link. Representing full or empty is now entirely in the links. We colored "link logic" grey, reserving the white center for the simplified joint.

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 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.

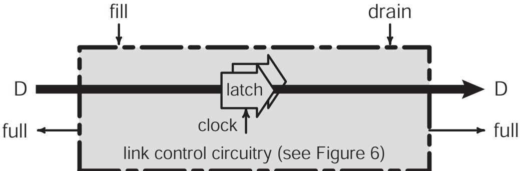


Figure 5 Naturalized link with standard link-joint interface. Wiring clock to $fill$ makes the latches normally-opaque, and wiring clock to $\neg full$ makes them normally-transparent.

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.

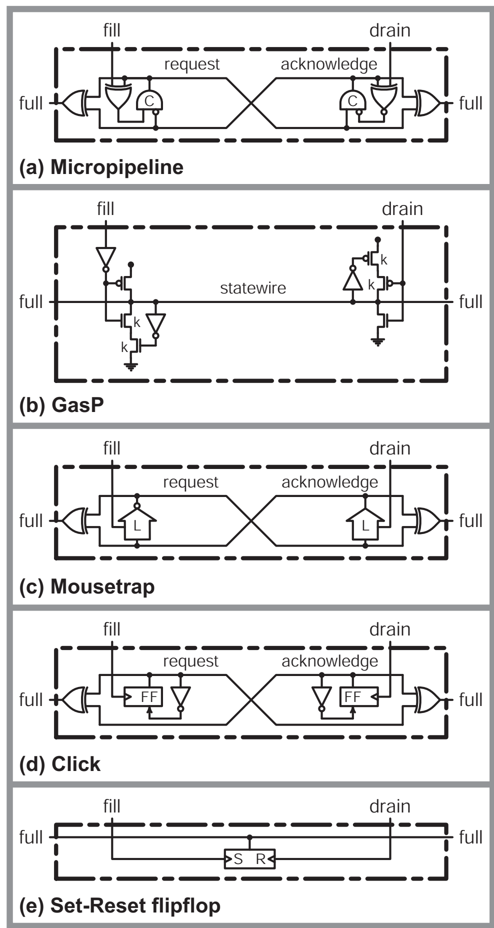


Figure 6 Implementation examples of link control circuitry. These supplement the link data circuitry of Figure 5.

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 $n = m = 1$.

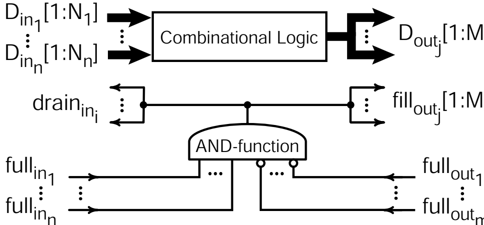


Figure 7 Design sketch of a joint without stored state with n naturalized incoming and m naturalized outgoing links, where $1 \leq i \leq n$ and $1 \leq j \leq m$. If data are just copied then $D_{out_j}[1:M]$ is the concatenation of $D_{in_1}[1:N_1]$ to $D_{in_n}[1:N_n]$ and $M = N_1 + \ldots + N_n$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

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.

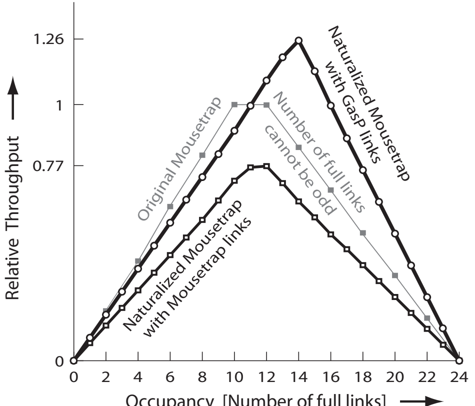


Figure 8 Three canopy graphs for simulations of rings with 24 pipeline stages. The center graph with fewer data points is from simulation with an original Mousetrap module. Naturalizing its links with the circuit of Figure 6(c) produces the lower graph. The increased number of logic gates costs performance. 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.

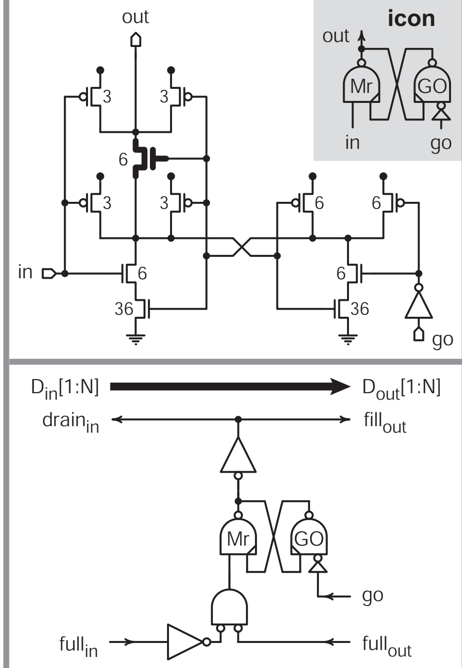


Figure 9 MrGO with its icon inset in the grey area (top), and a joint with MrGO (bottom). The bold central transistor in MrGO delays active-low grant signal, *out*, by conducting only after metastability ends. Transistor sizing reduces the logical effort from *in* to *out*. Split pull-up transistors 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
⋮	[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
⋮	[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]
⋮															
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]
⋮															
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.

Appendix E

Test Setup: Scan Chain Organization

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapters 5 and 6 and can be found as reference [33] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland.
  GasP Scan Chain Implementation for Init, Go, and Single Step. *Technical Report, ARC2013-smg08, Asynchronous Research Center, Portland State University*, December, 2013.

Asynchronous Research Center
Portland State University

Subject: **GasP Scan Chain Implementation for Init, Go, and Single Step**
Date: **3 December 2013**
From: **Swetha Mettala Gilla, Marly Roncken and Ivan Sutherland**
ARC#: **2013-smg08**

References:

[1] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. The Master Clear Blunder. *ARC2012-is21, Asynchronous Research Center, Portland State University,* 2013.
[2] — Init, Go and Single Step. *ARC2012-is04, Ibidem.*
[3] — Scan Testing GasP using a JTAG Controller. *ARC2013-smg09, Ibidem.*
Note: ARC2013-smg09 is included as Appendix F in Swetha's Ph.D. thesis.
[4] Steve Rubin, Electric VLSI Design System, *Static Free Software at http://www.staticfreesoft.com/.*

ABSTRACT

This ARC report presents the scan chain design and operation details to initialize, start, and single-step a given GasP design. Initialization and start bring the control part of a GasP design into a well-defined state so the design can operate in either normal mode, i.e., self-timed mode, or in test mode, i.e., single-step mode. This report does not cover the scan features for the datapath nor scan design features we may need for datapath parts that drive control circuitry or, vice versa, control parts that drive datapath circuitry.

Acknowledgements: we gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD thesis.

Table of Contents

1. Introduction 2
2. Scan Designs 3
2.1. GasP FIFO Design 3
2.2. Scan Chain Designs 6
3. Scan Operations 10
3.1. Initialization 10
3.2. Start Normal-Mode Operation 11
3.3. Single-Step Operation 12
4. Conclusion 13
5. APPENDIX 14

This document contains information developed at the Asynchronous Research Center at Portland State University. Disclose this information to whomever you choose. Distribution is permitted with adequate reference to the source of the ideas and information. You may reproduce this material for any educational use. Copies of the material must contain this notice.

1. Introduction

We seek to control and observe the state of every state-holding element on a chip, so we can test and gain confidence in the chip's functionality. It is common to use a scan chain for this purpose. The scan chain is a serial shift register that can provide access, over time, to many places on the chip. Its backbone is composed of two latches per bit or one flip-flop per bit.

The scan chain in commercial chips sometimes uses the flip-flops or latches of the working circuit as part of the scan chain. This reduces the total number of latches on the chip at the cost of making the scan chain an intimate part of the working circuits of the chip. This may cause two instances of the same working part of a chip to differ solely because the sequence of scan through their flip-flops differs. Tarik Ono at Oracle had trouble getting a modular FIFO accepted precisely because it differed in every instance solely because of scan chain layout.

We want to connect every state holding element to a scan chain, but avoid close coupling between the working parts of the chip and the scan means. It's essential that the scan chain be able to sense and report the state of each and every state-holding element in the chip. It is desirable that the scan chain also be able to deliver a value to each and every state-holding element in the chip.

To avoid close coupling between the main design and the scan design, we make all main latches have two inputs. One input will be devoted to the functional need of the object circuit. The other input will be connected to the scan chain. Thus, from the scan chain one can set a chosen value to the input of each and every such two-input latch.

This report focuses on the scan details to initialize, start, and single-step GasP control modules that are not data driven [1,2]. The additional scan design features of data-driven control modules are discussed in a later report.

Figure 1 shows a block diagram for the scan chain organization to initialize, start, and single-step a GasP design that's not data-driven. As reference example, we use a simple linear GasP FIFO consisting of three GasP storage modules. The configuration in **Figure 1** enables us to control and observe the voltage level of each module's start signal, go, and the voltage level of each statewire and keeper pair, sw-kp. The scan chain operations are programmed from the JTAG Box, using a predefined instruction set (see [3] for details).

The remainder of this report is organized as follows. Section 2 gives the design details of (1) the GasP FIFO to show which signals are controlled and observed by scan, and (2) the two scan chains in **Figure 1**. Section 3 explains how the given scan designs enable us to control and observe the go and sw-kp signals in the FIFO. Section 4 summarizes and concludes this report. Section 5 is an appendix with further schematic details for sub-cells used in the scan chain designs in Section 2.

NOTE: All designs in this report are in library Electric_180nm_ARC2013-smg08-09.

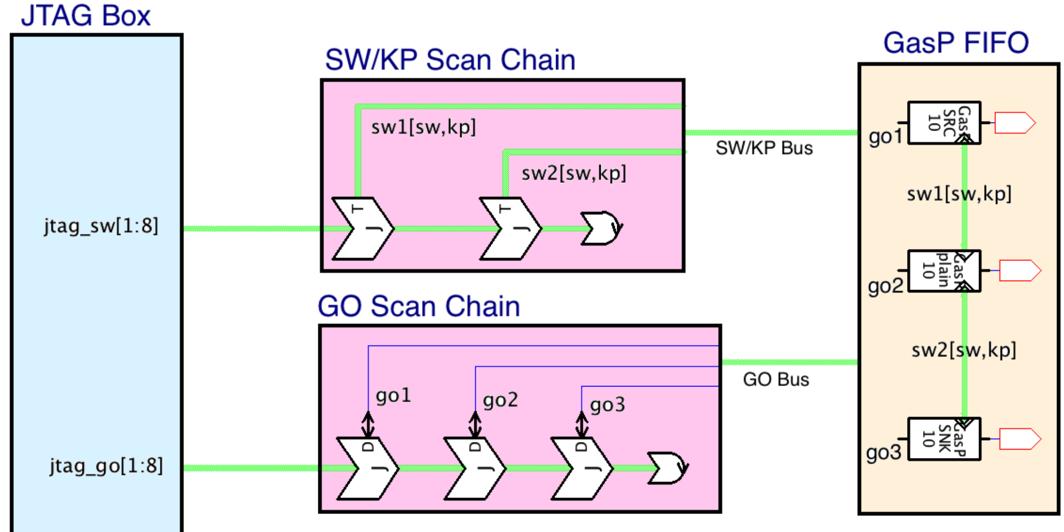


Figure 1: Block diagram to initialize, start, and single-step a GasP design that's not data-driven.

2. Scan Designs

This section gives the schematic design details of the scan chains and scan connections for the GasP FIFO used in the block diagram in **Figure 1**.

2.1. GasP FIFO Design

Figure 2 outlines the three-stage GasP FIFO design that we use as reference example. It uses three types of storage modules: SRC, plain, and SNK.

The schematic details of the plain version follow in Figure 3. It has the usual GasP Pred and Succ drivers, whose designs follow in **Figure 4**, for completeness sake. The difference between the SRC, plain, and SNK storage types is as follows. In SRC, the Pred driver is deleted and replaced by a wire connection from VDD to pred[sw]. Similarly, in SNK, the Succ driver is deleted and replaced by a wire connection from VSS to succ[sw].

The key signals for scan in the plain storage module are: go, pred[sw], succ[sw], pred[kp] and succ[kp]. The go signal is used to cleanly stop the module operation [1]. Wires pred[sw] and succ[sw] are bi-directional statewires, used for handshake communication between modules. In this schematics, state-wire HI means full or request, and LO means empty or acknowledge. Uni-directional signals pred[kp] and succ[kp] are used to turn off the keepers during initialization to prevent them from fighting the initial value settings.

NOTE: The schematics in this report show the signal directionalities for normal-mode operation. But signals go, pred[sw], succ[sw], pred[kp], and succ[kp] are bi-directional, i.e., they can be written (controlled) and read (observed), for scan test purposes.

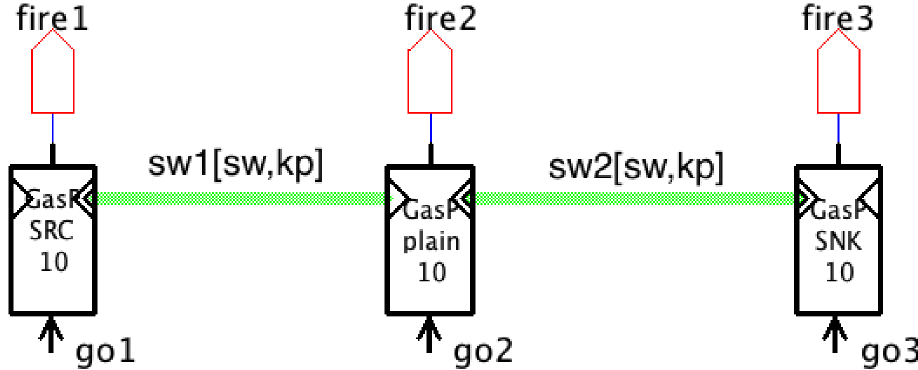


Figure 2: Three-stage GasP FIFO design, with three storage elements of type SRC, plain, and SNK.

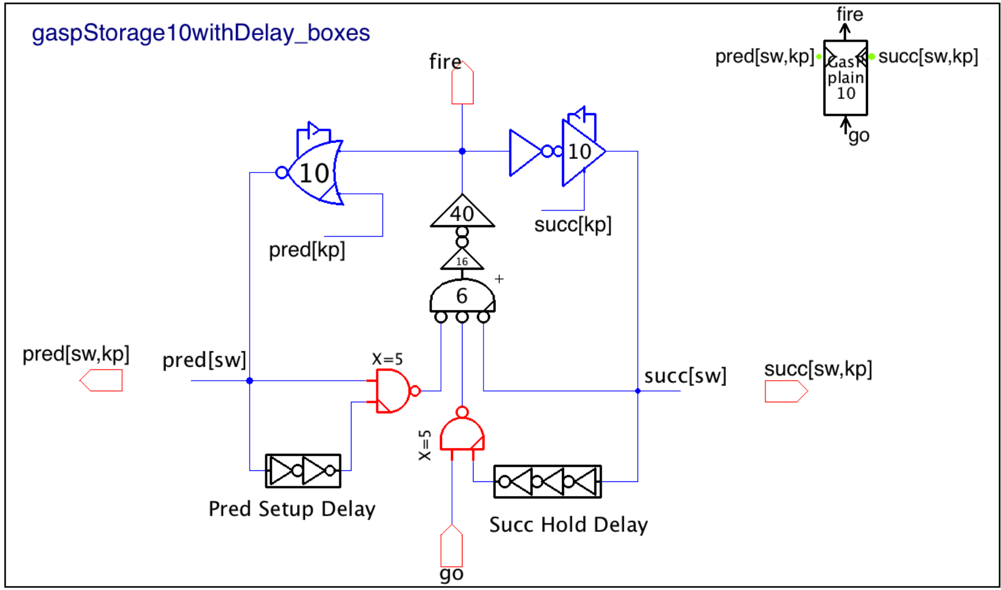


Figure 3: Schematic diagram and icon for a plain GasP Storage module. Our scan design can control and observe signals go, pred[sw], succ[sw], pred[kp] and succ[kp]. The rectangular gates named “Pred Setup Delay” and “Succ Hold Delay” are variable delay insertion points which can be programmed for timing closure.This design can be found in sub-library gaspL_go_init under the name gaspStorage10wDelay_boxes.

NOTE: For future fault-coverage experiments, we’ll likely replace the complex gates by single-inversion gates.

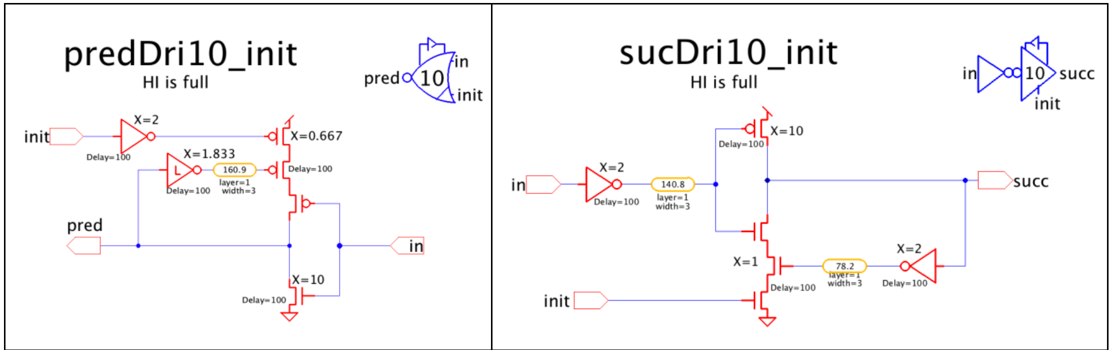


Figure 4: Schematics and icons for Pred Driver (left) and Succ Driver (right). Note the name change: signal init here is renamed to pred[kp] respectively succ[kp] in the Storage Pred and Succ Driver instances above. These two designs can be found under sub-library driversL_init, as predDri10wMC_init respectively sucDri10_init.

The normal-mode storage operation has two phases:

1. **Initialization:**
   Using the scan entry points to the Storage modules, we set all go signals LO, so the sw are no longer driven from the GasP modules; we may still have keeper fights, though. Then we set the sw signals to their intended value, and when we do so, the keepers are turned off automatically by the scan chain design for sw-kp pairs. For the FIFO example, we'll set all sw signals LO. After the initial state is propagated and stable, we can start the self-timed operation. To do this, we set all go signals HI.

2. **Self-Timed Operation:**
   In the example, we now have a stable initial state, with HI go signals and LO sw signals. As soon as pred[sw] goes HI, the plain storage module produces a HI transition on the fire signal, which results in three actions: the data-path latches capture the incoming data (absent in our example), succ[sw] is driven HI thus acknowledging and completing the 2-phase incoming handshake, and pred[sw] is driven LO thus acknowledging and starting a 2-phase outgoing handshake. Meanwhile, the self-resetting loops reset the fire signal to LO after five gate delays, ending both the data-path capture phase and the state-wire drives.

2.2. Scan Chain Designs

Figure 5 shows a more detailed version of the scan chain designs for go and sw-kp pairs. Each scan chain is a serially connected chain of **scan capture-launch-shift** cells terminated by a **scan cap** cell, and connected by an 8-bit daisy chain.

Each **Scan Capture-launch-shift** cell can:

- Capture, or read, data in parallel from the design into the scan chain.
- Launch, or write, data in parallel from the scan chain into the design.
- Shift data serially in and out from its incoming chain to its outgoing bus connection.

Each **Scan Cap** cell can:

- Turn the final scan data and far-out scan clock signals back in reverse-shift direction to the JTAG Box for scan output inspection and completion detection of the current scan shift cycle.

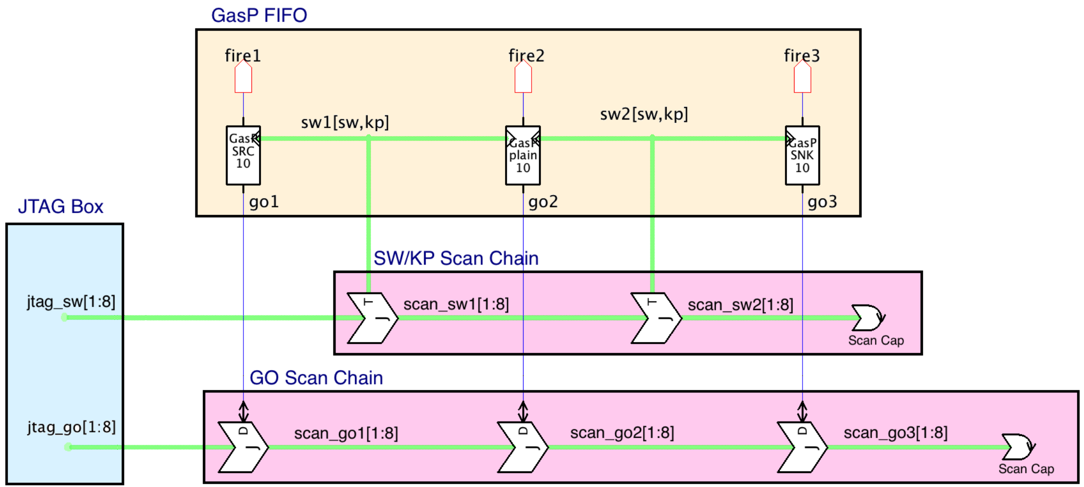


Figure 5: More detailed version of Figure 1, showing the scan connections between the JTAG Box and the GasP FIFO. We have separate scan chains for controlling and observing go and sw-kp signals. In particular:

- go scan signals can make each individual go signal high or low
- sw scan signals can make each individual statewire (temporarily) high or low.

Each scan chain design is a serially connected chain of capture-launch-shift cells terminated by a scan cap, and connected by an 8-bit bus with scan data and control signals.

The scan cap cells for the three scan chain designs are identical and designed à la **Figure 6**. Note that the two scan clock signals coming in from the bus as s[2] and s[3] and the single data signal, coming in from the bus as s[1], are returned to the bus as s[7], s[6], and s[8], respectively.

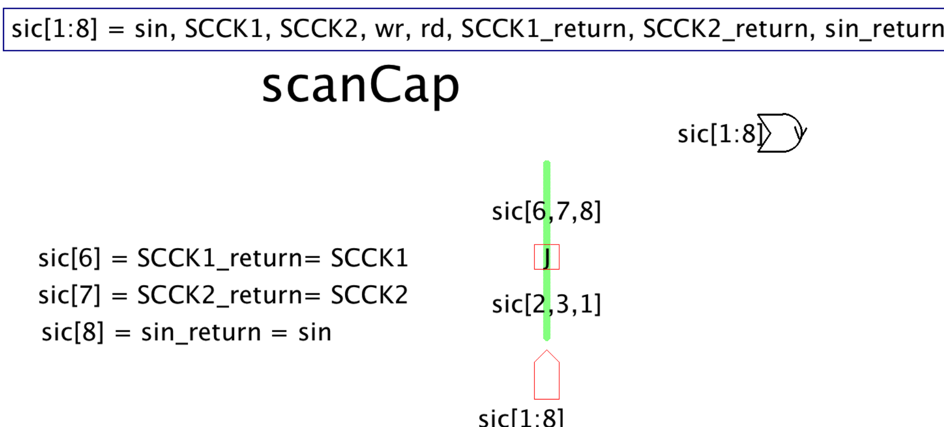


Figure 6: Schematic diagram and icon of the Scan Cap design. See sub-library scanL, scanCap.

The **Scan Capture-launch-shift** cells come in two flavors: one version stores the launched value on the signal in the design, and the other version shortly drives the launched value but then releases its drive to enable the design to take over and control the signal value. We use the "drive-and-keep" version to scan-control go signals, and the "drive-and-release" version to scan-control sw-kp signal pairs.

Both versions use a common structure to capture and shift scan and design values into the scan chain. We dubbed this common structure: Basic Scan Cell. Its design follows in **Figure 7**. The cells tagged "5A" are single non-inverting latch designs. For completeness sake, their schematics can be found in the Appendix of this report.

The complete **Scan Capture-launch-shift** versions follow in **Figure 8** and **Figure 9**(a-b). The tristate buffer design in **Figure 9**(a-b) replaces the latch design in **Figure 8**, and can be found in the Appendix of this report.

The first design in **Figure 9**(a), scanJT1, is simpler and works for the scan configuration in Figure 5. The second design in **Figure 9**(b), scanJT2, also works for scan configurations with mixed go and sw-kp scan cells in one long scan chain, because it can locally disable the wr signals for any sw-kp pair in the design. For our example, we will use scanJT1.

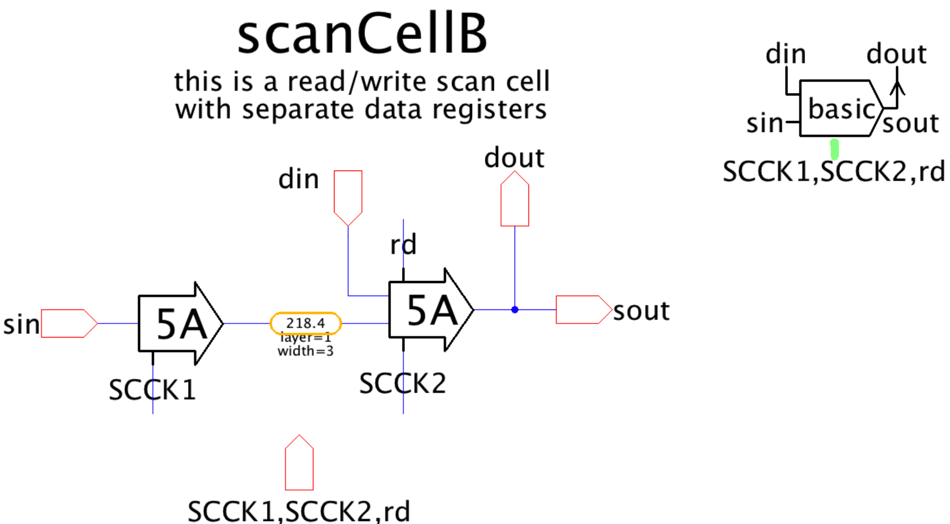


Figure 7: Basic Scan Cell design and icon to capture scan and design data and shift the data through the scan chain. This diagram is stored in sub-library scanL under the name scanCellB.

sic[1:8] = sin, SCCK1, SCCK2, wr, rd, SCCK1_return, SCCK2_return, sin_return

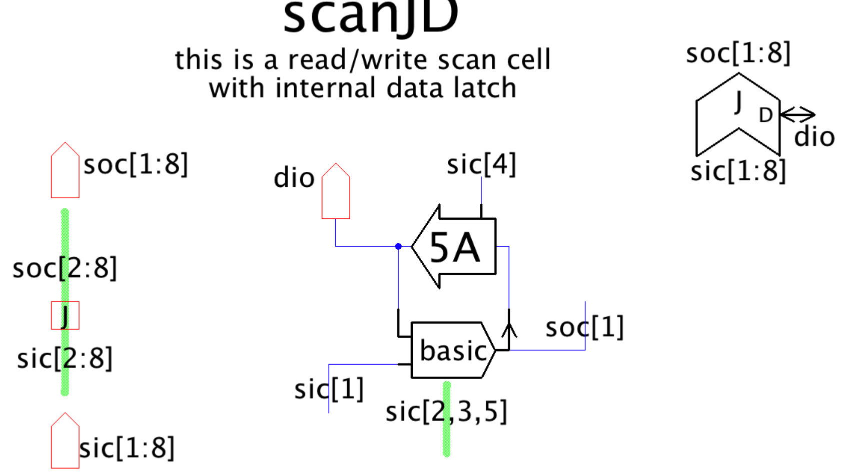
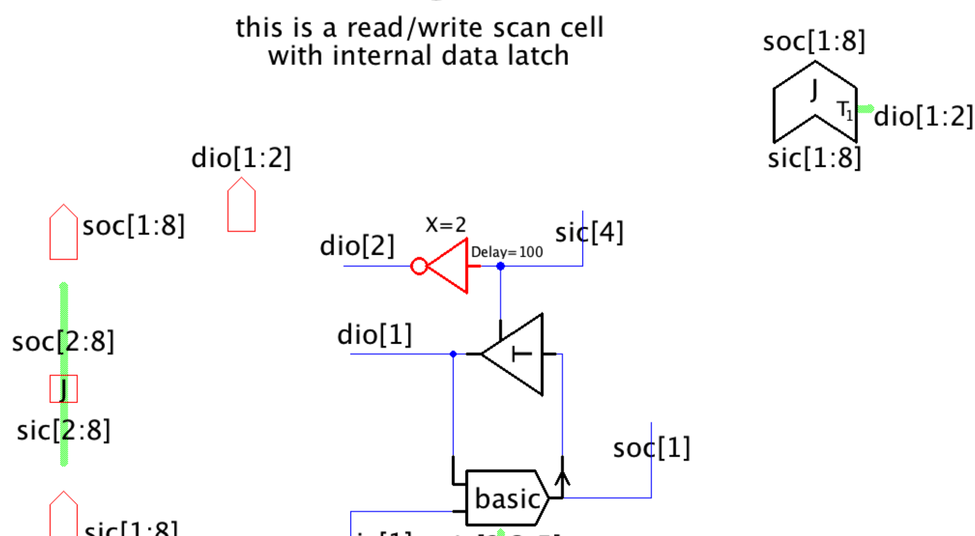


Figure 8: Schematic and icon for the “Drive-and-keep” **Scan Capture-launch-shift** version in the go scan chain in Figure 5. This diagram is stored in sub-library scanL under the name scanJD.

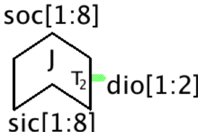
sic[1:8] = sin, SCCK1, SCCK2, wr, rd, SCCK1_return, SCCK2_return, sin_return

scanJT1



scanJT2

this is a read/write scan cell
with internal data latch



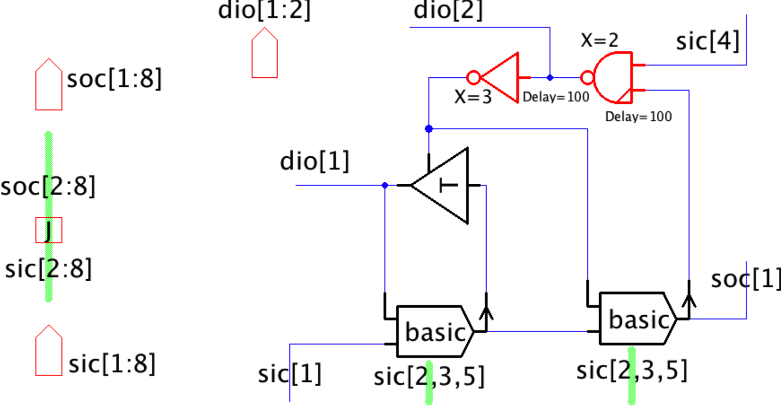


Figure 9: Schematic and icon for two possible "Drive-and-release" **Scan Capture-launch-shift** cell versions, that can be used for the sw-kp scan chain in Figure 5. They are stored in sub-library scanL under the names scanJT1 (top) and scanJT2 (bottom). The top design, scanJT1, is simpler and works for the scan configuration in Figure 5. The bottom design, scanJT2, also works for scan configurations with mixed go and sw-kp scan cells in one long scan chain, because it can locally disable the wr signals for any sw-kp pair in the design.

3. Scan Operations

The scan design in Section 2 plays a role in both the normal-mode operations of the design and in the test-mode operations. We use scan to initialize the design, i.e., to set the design in a well-defined state. A proper initial state is what's required prior to any normal-mode operation as well as prior to any test-mode operation. We also use scan to start the circuit in a delay-insensitive, i.e., well-defined and well-controlled, manner. Both normal-mode and test-mode operations require a delay-insensitive start state. It's the initial and start state settings that determine the difference between normal and test mode operations, but the control mechanisms to get these settings are the same.

Below, we explain how we control the settings of initial and start states, and how we can observe them for test validation purposes.

3.1. Initialization

The number one priority when initializing a GasP design is quickly to remove drive conflicts on the statewires due to an arbitrary setting that the circuit arrived in after it was powered up. We use an initialization procedure with the following step sequence:

- *Step 1:* For each scan chain, make the JTAG scan input signal, sin, LO.
- *Step 2:* For each scan chain, make both scan shift clocks, SCCK1 and SCCK2, HI.
- *Step 3:* For each scan chain, write the LO values to the go and paired sw-kp signals.

Note that his initialization procedure also sets each scan chain in a well-defined and stable initial state. Note also that instead of using overlapping SCCK1 and SCCK2 clocks in Step 2, we could have used alternate HI pulses on SCCK1 and SCCK2 to serially shift the LO sin value into each and every scan location.

The JTAG Box operates one scan chain at a time, and so we must serialize Step 1 to 3. It seems most logical to use the following ordering:

- **Init go:**
  First execute Step 1 to Step 3 for the go scan chain, because this will stop the stronger drives on the statewires by the Pred and Succ Drivers. For Step 2, the crux is to wait long enough for the LO sin value to propagate through the entire go scan chain. At that point, we can lower the scan shift clocks. For Step 3 we use a HI clock pulse on wr to drive-and-keep the LO scan input values onto the go signals.
- **Init sw-kp pairs:**
  Execute Step 1 to Step 3 for the sw scan chain, in order to create a well-defined initial state with all statewires LO. The Drive-and-release scan design version in **Figure 9**(a), i.e., scanJT1, automatically disables the keepers when wr is HI and the sw is driven LO, and it automatically enables the keepers when wr is LO.

The JTAG Box comes with an instruction set to program and issue the above sequence of steps. Details on this instruction set and on how it's used to run the above action sequence for initializing the 3-stage FIFO design can be found in ARC report [3]. For now, we'll assume that we can give the proper scan values to the JTAG bus interface between the JTAG Box and two scan chains. Each scan chain has its own 8-bit bus interface, sic[1:8], where sic is either jtag_go or jtag_sw, where:

- sic[1:8] = sin, SCCK1, SCCK2, wr, rd, SCCK1_return, SCCK2_return, sin_return
  - sin: the scan-data input to the scan chain.
  - SCCK1, SCCK2: are the scan clock signals for shifting data in and out.
  - rd: the clock-enable signal for reading data from the design into the scan chain.
  - wr: the clock-enable signal for writing data from the chain into the design.
  - SCCK1_return, SCCK2_return and sin_return: Scan Cap return signals.

Using the instruction set in [3,4], we can program the bus interfaces between the JTAG Box and the design and thus initialize the design as shown below in **Figure 10**.

Scan Interface	Scan data [1]	SCCK1 [2]	SCCK2 [3]	write [4]	read [5]	return bus [6:8]
1. jtag_go [1:8]	LO	HI	HI	LO	LO	wait long enough for completion with stable: [8]=[1]=LO [7]=[2]=HI [6]=[3]=HI
2. jtag_go [1:8]	LO	LO	LO	HI	LO	N.A.
3. jtag_go [1:8]	LO	LO	LO	LO	LO	N.A.
4. jtag_sw [1:8]	Repeat jtag_go[1:8] steps 1-3, but now for jtag_sw[1:8]					

Figure 10: Initialization instruction sequence at the bus interfaces between the JTAG Box and scan chains.

3.2. Start Normal-Mode Operation

The normal-mode operation requires a different initial state than the one obtained at the end of Step 4 in **Figure 10**. For instance, we'd like an initial start state for the 3-stage FIFO in **Figure 1**, where all go signals are HI, for self-timed operation, and all sw signals are LO, to start with an empty FIFO. We can create this start state by continuing the initialization sequence in **Figure 10** with the event sequence shown in **Figure 11**. In general, any go, and sw pattern can be obtained for the start state, using the scan shift capabilities of the scan chain.

Scan Interface	Scan data [1]	SCCK1 [2]	SCCK2 [3]	write [4]	read [5]	return bus [6:8]
1. jtag_go [1:8]	HI	HI	HI	LO	LO	wait long enough for completion with stable: [8]=[1]=HI [7]=[2]=HI [6]=[3]=HI
2. jtag_go [1:8]	HI	LO	LO	HI	LO	N.A.
3. jtag_go [1:8]	HI	LO	LO	LO	LO	N.A.

Figure 11: Post-init normal-mode start instruction sequence at the JTAG Box to scan chain interface.

3.3. Single-Step Operation

Single-step operation is really a bounded-step operation to test certain behaviors and possible fault scenarios in the design. It usually takes many single-step operations to adequately test the design. In single-step operation mode, we allow the design to run in limited self-timed fashion.

Figure **12** shows an example of a single-step run starting from go1 and go2 HI, go3 LO, and all sw signals LO. A correct design will transform this start state into an end state with sw1 and sw2 HI – which we can then scan out for inspection.
Note that we could have shifted in sw values serially, which would allow us to scan in new values for the single-step test we're preparing, while shifting out older sw values that can be used for test observation. Vice versa: while we shift out older values for inspection, we can shift in new values for the next single-step test.

Scan Interface	Scan data [1]	SCCK1 [2]	SCCK2 [3]	write [4]	read [5]	return bus [6:8]
1. jtag_go [1:8]	Repeat jtag_go[1:8] steps 1-3 in Figure 10					
2. jtag_sw [1:8]	Repeat steps 1-3 in Figure 10, but now for jtag_sw[1:8]					
3. jtag_go [1-8]	LO	LO	LO	LO	LO	N.A.
4. jtag_go [1-8]	LO	HI	LO	LO	LO	N.A.
5. jtag_go [1-8]	LO	LO	LO	LO	LO	N.A.
6. jtag_go[1-8]	LO	LO	HI	LO	LO	N.A.
7. jtag_go [1-8]	LO	LO	LO	LO	LO	N.A.
8. jtag_go [1-8]	Repeat jtag_sw[1:8] steps 3-7 above, but now with Scan data, i.e., jtag_go[1], set to HI					
9. jtag_go [1-8]	Repeat step 8 above					
10. jtag_go [1-8]	HI	LO	LO	HI	LO	N.A.
11. jtag_go[1-8]	HI	LO	LO	LO	LO	N.A.

Figure 12: Single-step instruction sequence at the JTAG Box to scan chain interface for the design in **Figure 1** and **Figure 5**, without the scan-out part. Step 1 guarantee that the statewires are not driven from the design, so we can safely drive them from the scan chain, in step 2.

4. Conclusion

This report presents a Scan Chain Setup for GasP designs, illustrated for a GasP FIFO module. The block diagram of the Scan Chain Setup is shown in Figure 1. It uses (1) a JTAG Box, (2) two scan chains to control and observe the go, and sw signals in the GasP design, and (3) the GasP FIFO design. We explained the organization of the two scan chains, and the scan operations.

Except for the time it takes to go from a power-up to a stable initial state, we can operate the scan design in a way that avoids drive fights and floating statewires.

We presented a single building block solution, scanJD, to control and observe go signals. We presented two possible scan building blocks, scanJT1 and scanJT2, to control and observe statewires. The example scan chain configuration of Figure 1 uses scanJD building blocks to scan go signals and can use either scanJT1 or scanJT2 building blocks to scan sw signals. Had we used a single-chain scan configuration in Figure 1, then that would have consisted of scanJD building blocks to scan go signals and only scanJT2 building blocks to scan sw signals. For single-chain scan configurations with mixed scan access to go and sw signals, we must be able to avoid driving any of the sw signals when we write the final scan values into the go signals, so as to ensure a delay-insensitive normal-mode or single-step test operation.

5. APPENDIX

For the sake of completeness, this section contains the implementations for the latch and tristate sub-designs used in the Basic, Drive, and Tri-State Scan Cell designs in Figure 7, Figure 8, and Figure 9. All designs are designed in Electric [4].

- **1-bit non-Inverting Latch:** This is a typical 1-input non-inverting latch design, with input bit in[1], high-enabled clock, hcl, and output bit out[1]. Its schematic design follows in **Figure 14**. Lower-level design details follow later in this appendix. The design operates as follows:
    - If hcl = HI then out[1] = in[1]
    - else out[1] keeps its current value
- **2-bit non-Inverting Latch:** A 2-bit multiplexed non-inverting latch design with inputs inA[1] and inB[1], non-overlapping clocks, hcl[A] and hcl[B], and output out[1]. Its schematic design follows in **Figure 14**. Lower-level design details follow later in this appendix. The design operates as follows:
    - If hcl[A] = HI (hcl[B] must be LO) then out[1] = inA[1]
    - elsif hcl[B] = HI (hcl[B] must be LO) then out[1] = inB[1]
    - else out[1] keeps its current value
- **1-bit Inverting Latch:** A typical 1-input inverting latch design, with input bit in[1], high-enabled clock, hcl, and output out[F]. Its design follows in Figure 15. Lower-level details follow later in the appendix. The design operates as follows:
    - If hcl = HI then out[F] = ¬in[1]
    - else out[F] keeps its current value
- **2-bit Inverting Latch:** A 2-bit multiplexed inverting latch design with inputs inA[1] and inB[1], non-overlapping clocks hcl[A] and hcl[B], and output out[1]. Its schematic design follows in **Figure 16**. Lower-level design details follow later in this appendix. The design operates as follows:
    - If hcl[A] = HI (hcl[B] must be LO) then out[1] = ¬inA[1]
    - elsif hcl[B] = HI (hcl[B] must be LO) then out[1] = ¬inB[1]
    - else out[1] keeps its current value
- **LatchPointF:** This is the basic building block for the 1 and 2-bit Inverting Latch designs. It has a 1-bit input in[1], two tristate outputs out[F] and out[T], and a high-enabled clock, hcl. The design follows in **Figure 17** and operates as follows:
    - If hcl = HI then out[F] = ¬in[1] and out[T] = in[1]
    - else out[F] and out[T] keep their current values
- **Keeper**: Back-to-back keeper design, with a strong inversion from out[s] to out[B] and a weak inversion from out[B] to out[s]. Its design follows in **Figure 18**. Note that the strongly driven side out[B] is the one connected to the output pin in the 1-bit and 2-bit Inverting Latch designs
- **Tristate Buffer:** Typical tri-state design solution for driving a 1-bit input in[1] onto the 1-bit output out[1] by making the high-enabled clock hcl HI, and then releasing the drive by lowering the clock. The schematic design follows in **Figure 19**.

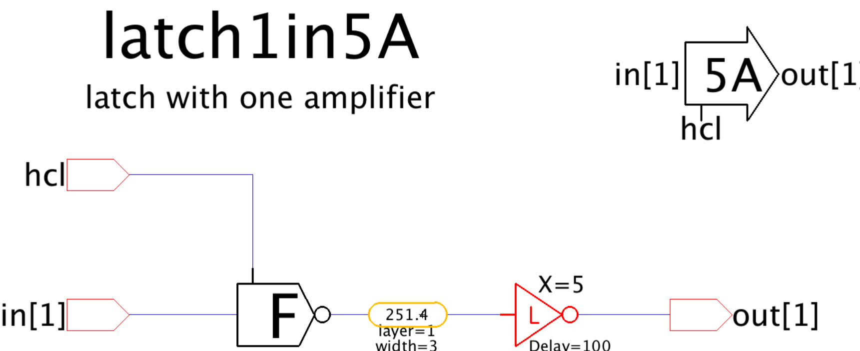


Figure 13: Schematic diagram and icon for the 1-bit non-Inverting Latch design. See latchesL, latch1in5A.

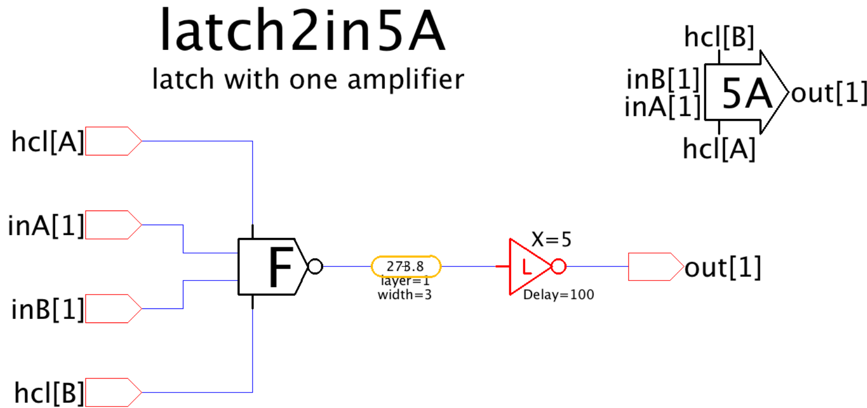


Figure 14: Schematic diagram and icon for the 2-bit non-Inverting Latch design. See latchesL latch2in5A.

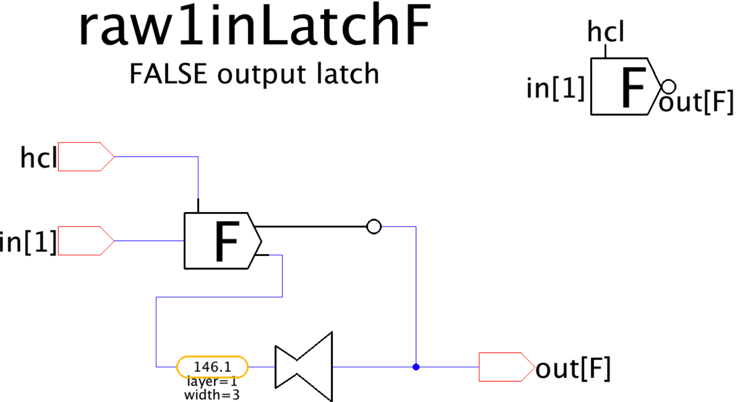


Figure 15: Schematic diagram and icon of a 1-bit Inverting Latch. See sub-library latchesL raw1inLatchF.

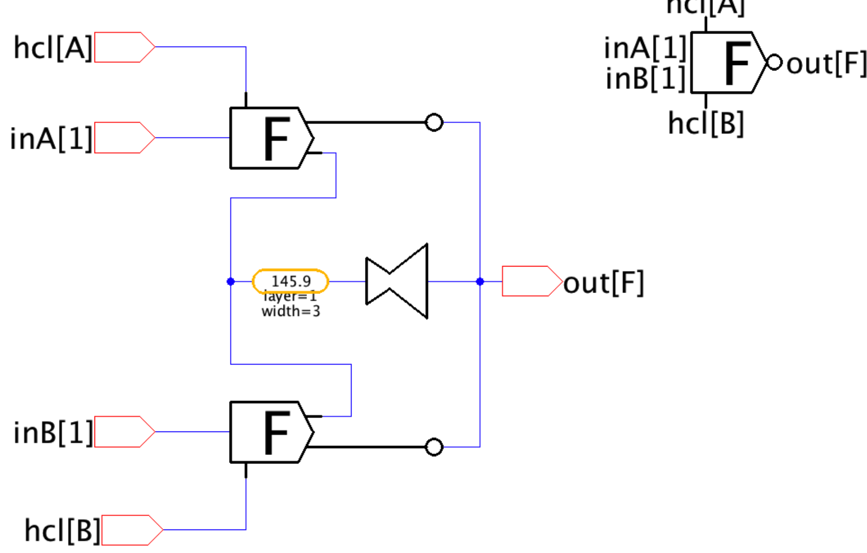


Figure 16: Schematic diagram and icon of the 2-bit Inverting Latch. See sub-library latchesL, raw2inLatchF.

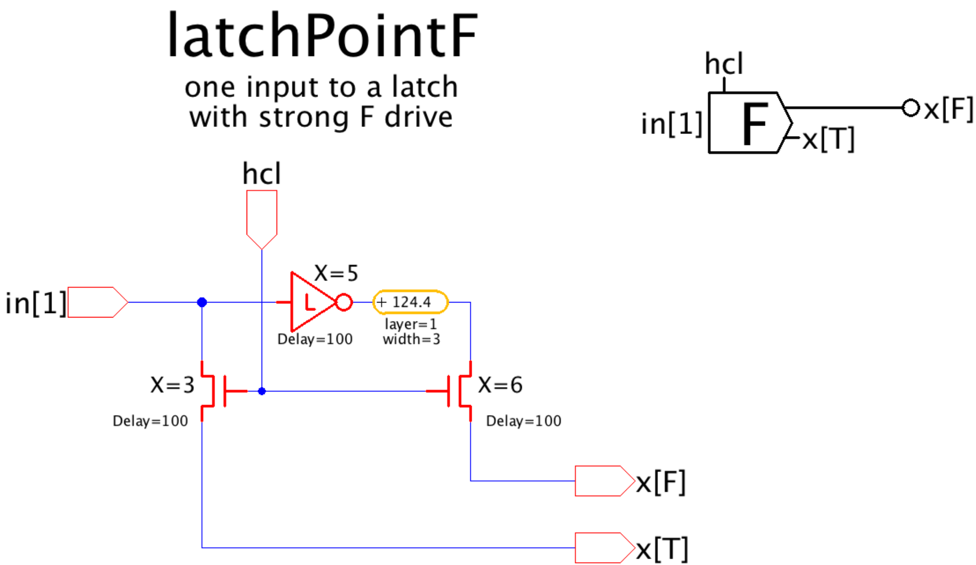


Figure 17: Schematic diagram of LatchPointF used as sub-design in the scan latch designs. The special inverter tagged "L" is a LO-threshold inverter, with an N drive strength twice the P drive strength. The icon for this design is in the top-right corner. This diagram is stored in sub-library latchesL under the name latchPointF.

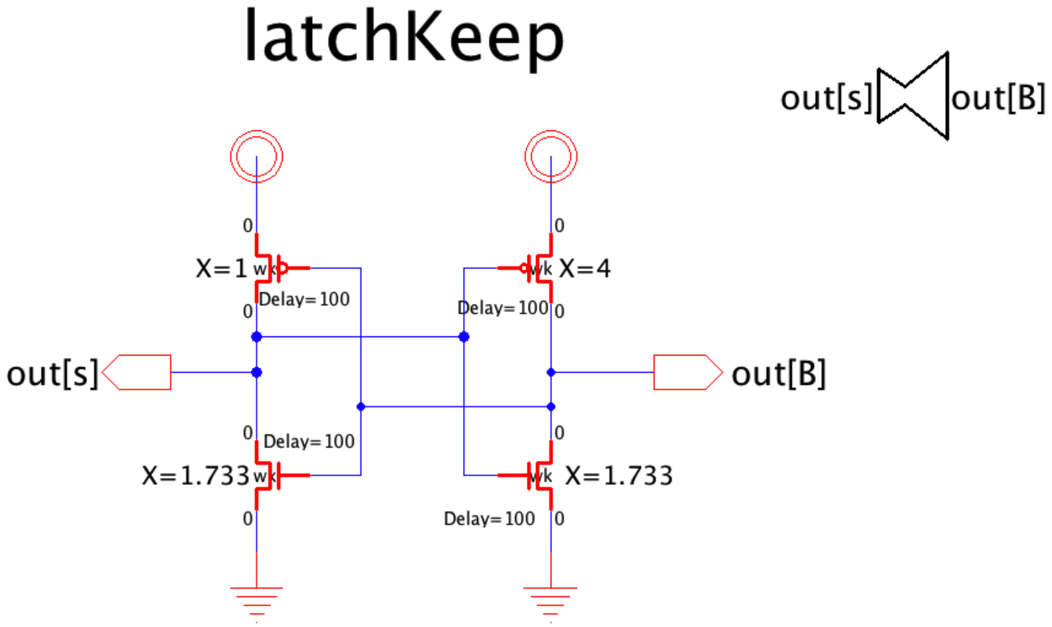


Figure 18: Schematic diagram and icon of the Keeper design. See sub-library latchPartsL under latchKeep.

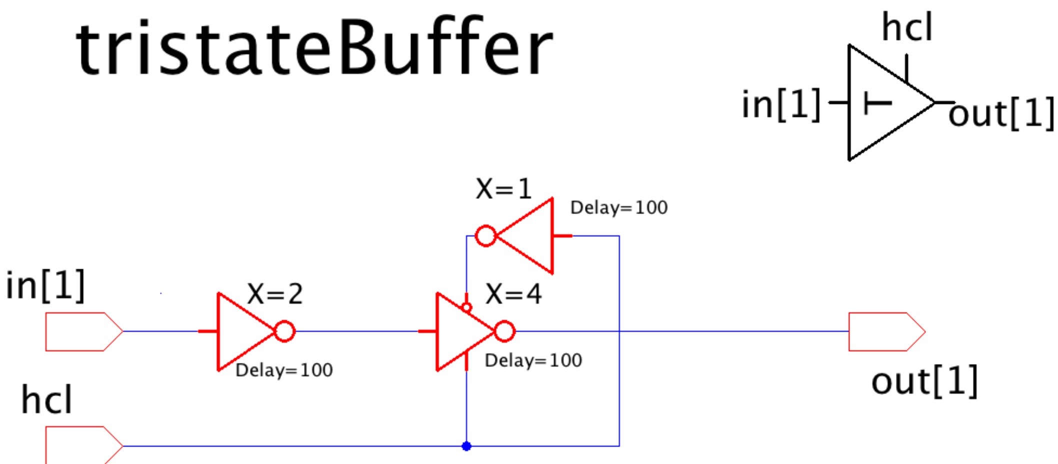


Figure 19: Tristate buffer design and icon, used in the two "Drive-and-release" **Scan Capture-launch-shift** cell versions. This diagram is stored in sub-library scanL under the name tristateBuffer.

Appendix F

Test Setup: JTAG Controller

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapters 5 and 6 and can be found as reference [34] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland.
  Scan Testing GasP using a JTAG Controller. *Technical Report, ARC2013-smg09, Asynchronous Research Center, Portland State University*, January, 2014.

Asynchronous Research Center
Portland State University

Subject: **Scan Testing GasP using a JTAG Controller**
Date: **10 January 2014**
From: **Swetha Mettala Gilla, Marly Roncken and Ivan Sutherland**
ARC#: **2013-smg09**

References:

[1] SML# 2008-xxxx The Infinity Chip, Sun Microsystems tested silicon, 2008.
[2] SML# 2009-xxxx The Marina Chip, Sun Microsystems tested silicon, 2009.
[3] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. The Master Clear Blunder, *ARC2012-is21, Asynchronous Research Center, Portland State University,* 2013.
[4] — Init, Go and Single Step, *ARC2012-is04, Ibidem.*
[5] — GasP Scan Chain Implementation for Init, Go, and Single Step, *ARC2013-smg08, Ibidem.*
Note: ARC2013-smg08 is included as Appendix E in Swetha's Ph.D. thesis.
[6] IEEE-SA Standards Board, IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-2001 (Revision of IEEE Std 1149.1-1990), *IEEE Computer Society*, 14 June 2011.
[7] Steve Rubin, Electric VLSI Design System, *Static Free Software at http://www.staticfreesoft.com/*.

Abstract

This document shows how we can use the Joint Test Action Group (JTAG) controller to scan test GasP circuits. We give the details of the Test Access Point (TAP) architecture and the JTAG controller implementation and its use in operating the scan chains for the GasP control logic introduced in [5]. This document also works out the details of some of the GasP scan experiments mentioned in [5]. Our JTAG solutions are designed in Electric [7] and tested using SPICE level simulations. We adapted them from the 90nm Sun Microsystems designs in [1,2] to a 180nm technology, extended them with an overlapping scan shift clock mode for fast initialization of the design under test, and we revised the GasP scan chain control logic to accommodate the Init, Go, and Single Step operations explained in [3,4,5].

All designs in this report are available for downloading using electric 9.0 for 180nm, using Swetha's Electric directory version Electric_180nm_ARC2013-smg08-09 with its anOpener.jelib opened as key library, and Swetha's JTAG_experiments2013.jelib opened as second Electric library.

Keywords: JTAG, TAP controller, IR, Pulse Setup, Scan Chains, Scan testing, GasP.

Acknowledgements: We gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD work.

This document contains information developed at the Asynchronous Research Center at Portland State University. Disclose this information to whomever you choose. Distribution is permitted with adequate reference to the source of the ideas and information. You may reproduce this material for any educational use. Copies of the material must contain this notice.

Table of Contents

1. INTRODUCTION 3
2. JTAG Box Overview 4
3. JTAG Control Module 7
3.1. TAP Controller 8
3.2. Instruction Register (IR) Logic 13
3.1.1 IR Control 13
3.1.2 Instruction Register (IR) – Shift Register Part 16
3.1.3 Instruction (IR) Decoder 18
3.3. TDO Driver 19
3.4. Output Logic 19
4. JTAG Central Module 20
4.1. Scan Control Group 20
5. JTAG Box Programming Interface 25
5.1. TCK and TRSTb Pulse Setup 25
5.2. TMS Pulse Setup 26
5.3. TDI Pulse Setup 29
6. Scan Test Experiments: Simulation Setup and Results 33
6.1. Programming the Input Pulse Generators 33
6.2. Scan Operations and SPICE Simulation Results 35
7. Summary 41

1. INTRODUCTION

The Joint Test Action Group (JTAG) IEEE 1149.1 standard is used to test an assembled product. This standard defines the test access port (TAP) and boundary-scan architecture that can be adopted as a standard feature of integrated circuit design. Boundary scan test originally focused on controlling the IO pins of each chip in order to allow testing the interconnections between chips on a board, whereas Scan Chain test was used to test the internal logic of the chip or design under test. With System-on-Chip architectures, the board is now the chip, and the chip is now an on-chip module or sub-system. And with that, the JTAG IEEE 1149.1 TAP architecture has become a standard interface for testing on-chip communication and on-chip systems. In this ARC report, our focus is on testing the GasP modules and handshake connections in a given design. In this report, our design can be seen as a single product rather than as an assembly of products. As such, we have no need for specific boundary scan test capabilities in the TAP controller, and will focus on plain scan test capabilities.

In [5], we explained our scan chain organization for testing the control modules in a GasP design. As reference design in [5] we used a three-stage GasP FIFO. In the present report, ARC2013-smg09, we will use use slightly larger FIFO with five stages, and we will go over some of the scan chain test experiment shown in [5], and show how we can set them up and program them through the chip JTAG controller interface. **Figure 1** shows a block diagram of the new test experiment. It is similar to the block diagrams in Figures 1 and 5 in [5], with the exception that we now use five rather than three stages for the linear FIFO and that we now focus on the JTAG Box.

The JTAG Box in **Figure 1** has three key modules: Pulse Setup, JTAG Control, and Scan Control Group. Modules JTAG Control and Scan Control Group are combined into a single module: JTAG Central. The functions for each of these modules are as follows:

1. **Pulse Setup**: This is our programming interface. It can reside either on or off chip. It allows us to define signal input pulses to the JTAG Central module, and group them into pre-defined building blocks to obtain a more abstract, more modular, more user-friendly, and less error-prone approach for programming our scan tests.
2. **JTAG Central**: This is the interface module between (1) our programming environment as defined by the Pulse Setup module, and (2) the scan chains with the GasP design under test as defined in [5]. It is partitioned into two sub-modules: JTAG Control and Scan Control Group.
3. **JTAG Control**: This module contains the IEEE 1149.1 standard TAP architecture. We extended the architecture with some extra instruction register control logic for special scan features that we like, such as non-overlapping shift clocks for fast initialization. This extension falls within the IEEE standards, but may not fall within the programming interface developed at the time by Sun Microsystems for [1,2]. So, to re-use the existing software, we may have to drop our special scan features.
4. **Scan Control Group**: This module groups the Test Data Input (TDI) signal coming from the Pulse Setup module with the signals generated by the TAP controller and IR logic from the JTAG Control module. It translates these into specific scan chain signal settings and hands a Test Data Output (TDO) signal back from the scan chains to the JTAG Control module.

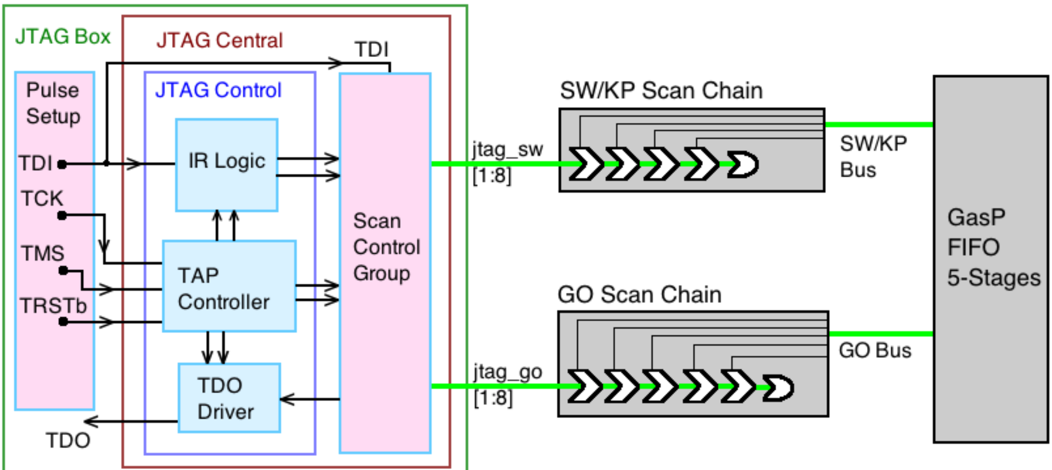


Figure 1: Experimental test setup with a JTAG Box (left), SW/KP and GO scan chains (middle) and a linear GasP FIFO with five stages acting as the design under test.

The rest of this report is organized as follows. Section 2 compares our JTAG Box design with the IEEE standard and the original implementation by Sun Microsystems, now Oracle, in [1,2]. Section 3 discusses the JTAG Control module, Section 4 discusses the JTAG Central module, and Section 5 gives the design details of the Scan Control Group. Section 6 explains the Pulse Setup module. Section 7 discusses the scan test experiments that we ran scan and presents their SPICE simulation results with waveforms. Section 8 concludes this report.

2. JTAG Box Overview

A more detailed JTAG Box implementation follows in **Figure 2**. It satisfies the IEEE standard in [6] and is based on the version implemented by Sun Microsystems, now Oracle, for [1,2]. The pink modules are the ones we added to the standard IEEE architecture. Similar additions are present in the Sun-Oracle version. They are:

1. A **Pulse Setup** module to provide a more abstract user-friendly programming interface, on- or off-chip, which overlaps with Sun-Oracle's implementation.
2. A **Scan Control Group** module for individual control for multiple scan chains, like the SW/KP and GO scan chains in **Figure 1**. The Sun-Oracle implementation also supports multiple scan chains.
3. A **Clock Generator** to generate non-overlapping clocks. Our design and test implementations use level-triggered single latches and double-latch master-slave flipflops [5] rather than edge-triggered flipflops popular in synchronous designs. For delay-insensitive operation, our latch-based designs use non-overlapping clocks: one clock, CK1, for the master latch, and another clock, CK2, for the slave latch. The clock generator in **Figure 2** takes the external Test Clock, TCK, and generates non-overlapping CK1 and CK2. Its design follows in **Figure 3**.
4. An **IR Control** module to overlap scan shift clocks for fast initialization of the design. This module is absent in Sun-Oracle's implementation.

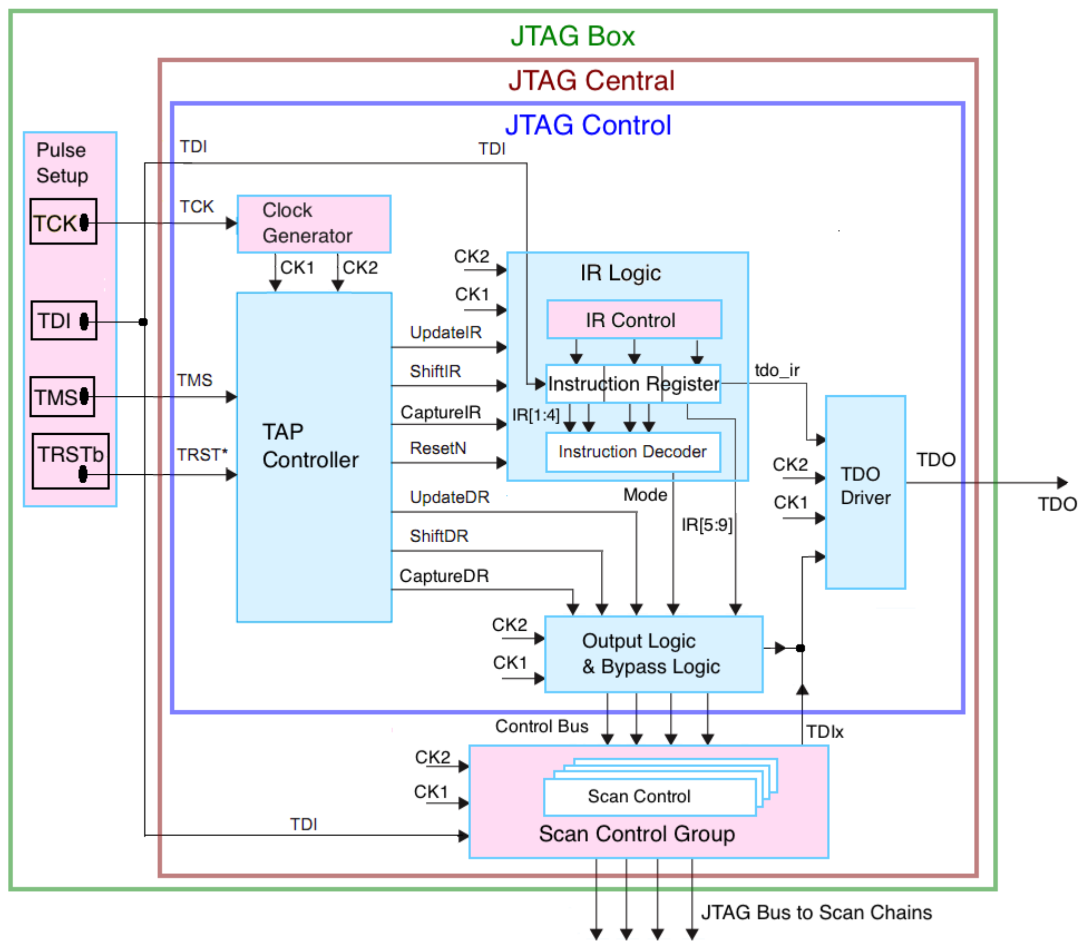


Figure 2: A more detailed version of the architecture of the JTAG Box design than the one given in **Figure 1**. The pink-colored modules are extensions to the IEEE standard given in [6], but compatible with the standard.

These pink-colored add-ons fall within the IEEE standards, but may not fall within the programming interface developed by Sun-Oracle, and used for testing the Infinity and Marina chips in [1,2]. If we are to re-use the existing software from Sun-Oracle, we may have to drop the fast initialization mode by overlapping shift clocks as supported by our IR Control module. The Sun-Oracle implementation provides fast initialization through a global master clear signal, mc, which is routed as part of its 9-bit scan bus. Our 8-bit scan bus matches Sun-Oracle's without the mc signal. Also our non-overlapping clock strategy is slightly different but we believe that our clocks are phase and skew consistent with the Sun-Oracle versions.

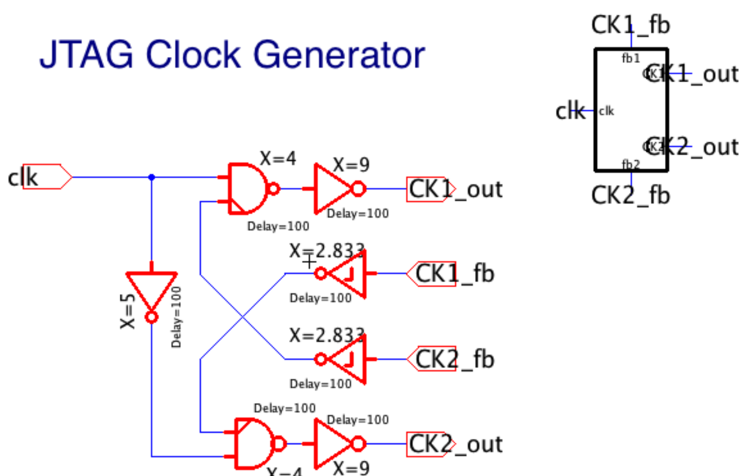


Figure 3: Implementation and icon for the JTAG Clock Generator in **Figure 2**, used to translate TCK into two non-overlapping clock signals, CK1 and CK2. CK1_fb and CK1_fb are delayed versions of CK1_out and CK2_out respectively, taken from the farthest point in the distribution network of CK1_out and CK2_out. By sensing CK1_fb and CK2_fb we can guarantee that we have non-overlapping CK1_out and CK2_out, independent of the delays in the two clock distribution networks. After stabilizing:

- clk HI results in both CK1_out and CK1_fb HI and both CK2_out and CK2_fb LO, and
- clk LO results in both CK1_out and CK1_fb LO, and both CK2_out and CK2_fb HI.

This design lives in Electric_180nm_ARC2013-smg08-09, jtagCentralSubModules_unchanged, as clockGen.

The clock interface constraints are set by the IEEE standard: The Test Data Input, TDI, must be set up shortly before the Test Clock, TCK, rises. The Test Data Output, TDO, becomes available upon TCK falling. We arrange the part in-between differently than Sun-Oracle:

- We re-use the top-level Sun-Oracle clock generator shown in **Figure 2**. But we add two lower-level clock generators in the IR Logic and Scan Control Group that, in addition to generating complementary clocks, allow both generated clocks to be (1) overlapping LO, and (2) overlapping HI.

- TAP Controller outputs become available at CK2 for us, at CK1 for Sun-Oracle; ditto for outputs from the IR Logic and Output & Bypass Logic, and for outputs from the Scan Control Group, and even for the scan chains themselves.

  We want to use CK1 as master clock and CK2 as slave clock for all master-slave flipflops in the test configuration. As a result, we pay an extra TCK clock cycle to generate TDO at the TDO module, one for CK1 followed by another one for CK2, as opposed to just CK2 for Sun-Oracle. For scan operations, this difference amounts to pretending there is one more scan register in the scan chain than there really is. This does not jeopardize the usability of the scan software.

In conclusion: apart from fast initialization, we are able to re-use the Sun-Oracle scan software to program GasP scan operations for SW/KP and GO scan chains.

3. JTAG Control Module

Figure 4 shows our implementation of the JTAG Control Module in Electric [7]. It includes the TCK to CK1-CK2 clock generator of **Figure 3** for driving our single latches and master-slave edge-triggered flipflops. It also shows the sub-modules in the IR Logic and in the Output Logic and Bypass Logic.

In the following sub-sections we'll go over the bigger sub-modules, and explain their role in scan testing GasP control circuits. We'll skip the Bypass sub-modules, because they are irrelevant for scan testing GasP.

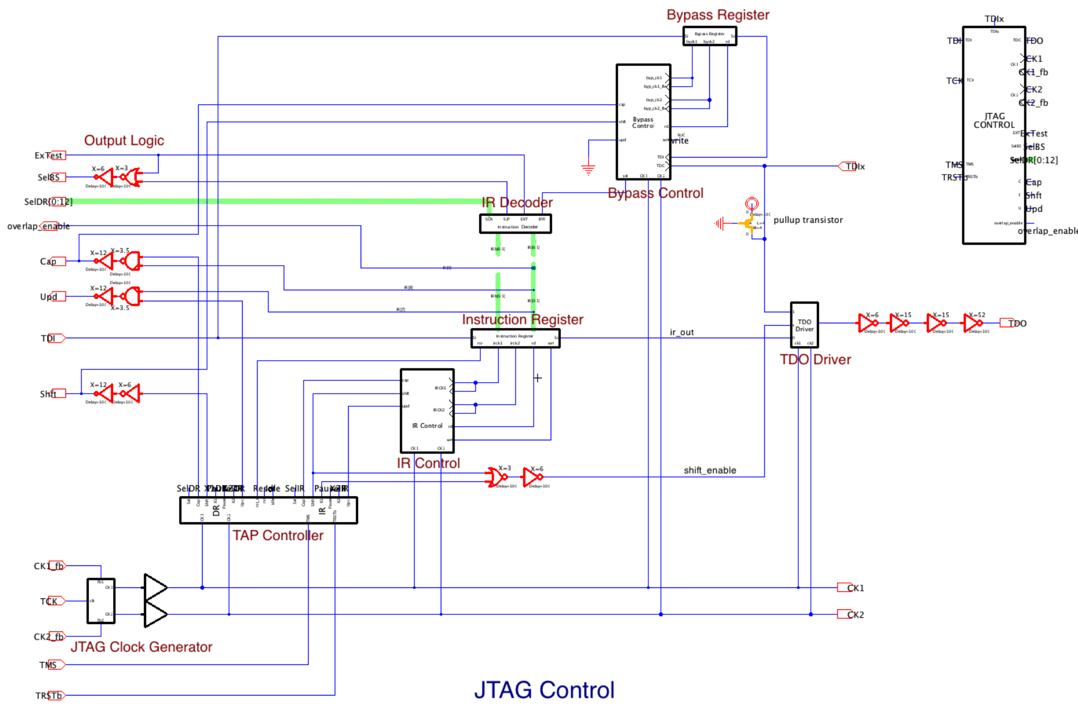


Figure 4: Schematic diagram and icon of our JTAG Control Module design in Electric [7]. This design is stored in library Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as jtagControlNew.

3.1. TAP Controller

The Test Access Port (TAP) Controller is implemented as a 16-state Finite State Machine, with states and transitions as depicted in **Figure 5**. It operates on three binary input signals, TCK, TMS, and TRST, where:

- **TCK**, for Test Clock, triggers the state-to-state transitions, with a HI pulse.
- **TMS**, for Test Mode, determines the state of the TAP controller. In our implementation, we read TMS at CK2 HI, which corresponds to TCK LO. So, basically, TMS is captured at TCK LO so it is stable by the next TCK HI pulse that triggers the next state transition in the TAP Controller. TMS must be stable some given setup time prior to the falling edge of TCK, and its new value must be updated in time before the next fall of TCK.
- **TRST**, for Test Reset, initializes the TAP controller (see also end of Section 3.1).

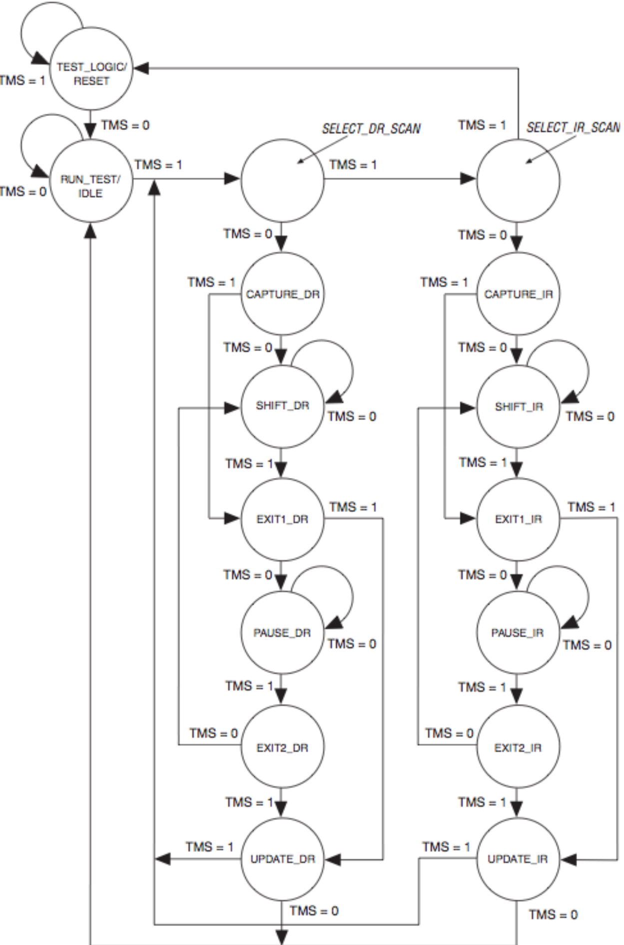


Figure 5: Finite State Machine implementation of the TAP Controller; see also Figure 6-1 in [6].

The states in the TAP Controller can be divided into four groups:

1. Reset
2. Idle
3. IR control (select, capture, shift, exit1, pause, exit2 and update)
4. DR control (select, capture, shift, exit1, pause, exit2 and update)

State groups 3 and 4 control the TAP Controller outputs: UpdateIR, ShiftIR, CaptureIR, ResetN, UpdateDR, ShiftDR, CaptureDR. In **Figure 5**, the states in groups 1 and 2 are in the top-left column. The states in group 3 for IR control are in the right column. The states in group 4 for DR control set are in the middle column. A brief description of the states follows next.

- TEST_LOGIC_RESET: Group 1 (Reset).
  This state is the intended initial TAP Controller state. It triggers ResetN to reset the IR Logic. All test logic is disabled in this state so that the normal-mode system operation can continue unhindered.

- RUN_TEST/IDLE: Group 2 (Idle).
  This is a stable TAP Controller state between scan operations, used for running a preset test or staying idle for testing while leaving the current instructions intact.

- SELECT_DR_SCAN and SELECT_IR_SCAN: Group 3 or 4 (IR or DR Control).
  These are intermediary states exited on the next TCK cycle, used to control instructions registers (IR) or to control data registers (DR) or to reset. In case of IR or DR control, we distinguish successor states *s*_x where x is DR or IR, and *s* is:
    - CAPTURE: this state triggers output CaptureDR respectively CaptureIR, which results in a parallel load operation from design data into the scan chain (a.k.a. “read”) respectively from IR data into the IR shift register.
    - SHIFT: this state triggers output ShiftDR respectively ShiftIR to serially shift the Instruction Register respectively scan chain by one position. SHIFT is entered and exited on TCK LO. While in SHIFT, each subsequent TCK HI transition produces a shift-by-one.
    - EXIT1 or EXIT2: temporary controller state, from which the current scanning process can be terminated.
    - PAUSE: Temporary suspension state that keeps existing settings.
    - UPDATE: state that triggers UpdateDR respectively UpdateIR, which results in a parallel load from scan data into the design (a.k.a. “write”) respectively from the IR shift register to IR’s output as the new instruction.

A typical test sequence involves clocking TCK at some cycle rate, setting TRST to 0 for a few cycles to reset the TAP controller states and the IR logic, returning TRST to 1, and then toggling or leaving TMS as needed per TCK cycle to traverse through the intended set of states. Note that a series of five consecutive CLK cycles with TMS 1 resets the system to the TEST_LOGIC_RESET state, from any state in **Figure 5**.

Figure 6 shows the design of the TAP controller as we implemented it in Electric. Note the structural similarity with the IEEE Finite State Machine diagram in **Figure 5**.
The key differences to note for our implementation are:

1. CK1 clocks the master latch of each so-called state latch, which is a flipflop, and CK2 clocks the slave latch. Signal TMS is clocked in through a single latch via CK2 at the top-left. This clocking scenario is similar to the Sun-Oracle version.
2. Output as well as internal state signals in the TAP Controller are taken from the slave latch of each state latch, while the Sun-Oracle version takes its outputs from the master and its internal signals from the slave. It is this difference that eventually culminated in the one clock cycle difference at the TDO pin of our versus the Sun-Oracle's JTAG Box design, as discussed at the end of Section 2.

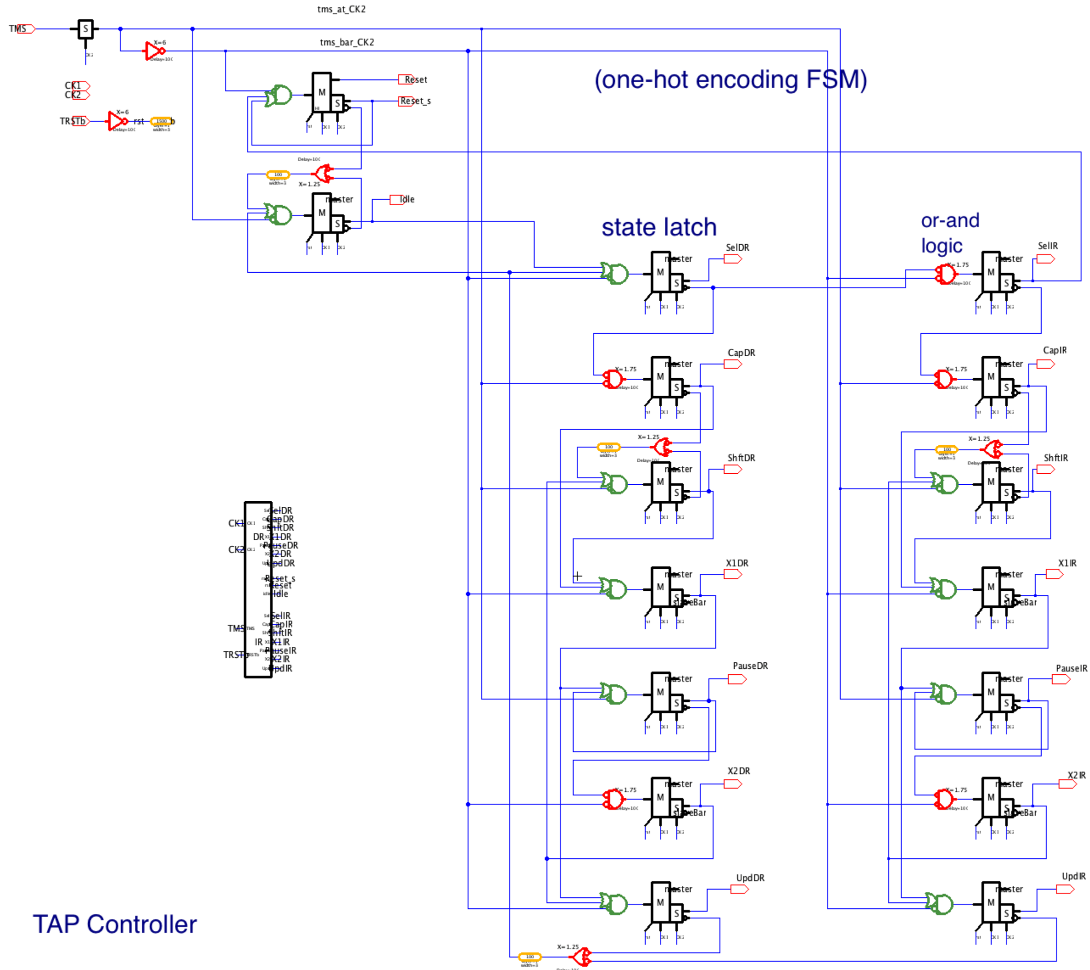


Figure 6: Schematic diagram and icon of our TAP controller module. This design is stored in subdirectory Electric_180nm_ARC2013-smg08-09, sub-library jtagCentralLatest2013, as tapCtlJKLNew.

The design of the master-slave state latch follows in **Figure 7**; it operates as follows:

- Always: slave = ~slaveBar.
- If rst = HI then master = LO
- If CK1 = HI then master = next
- If CK2 = HI then slave = master

For proper operation, signals CK1 and rst are never HI at the same time. For the TAP Controller, this translates to: TCK and TRST are never HI at the same time. Though there is no rule in the IEEE specification that requires this, Section 4.6 in [6] gives some recommendations to ensure deterministic operation of the test logic for changing TRST* (either TRST or its complement TRSTb) in relation to TCK and TMS.

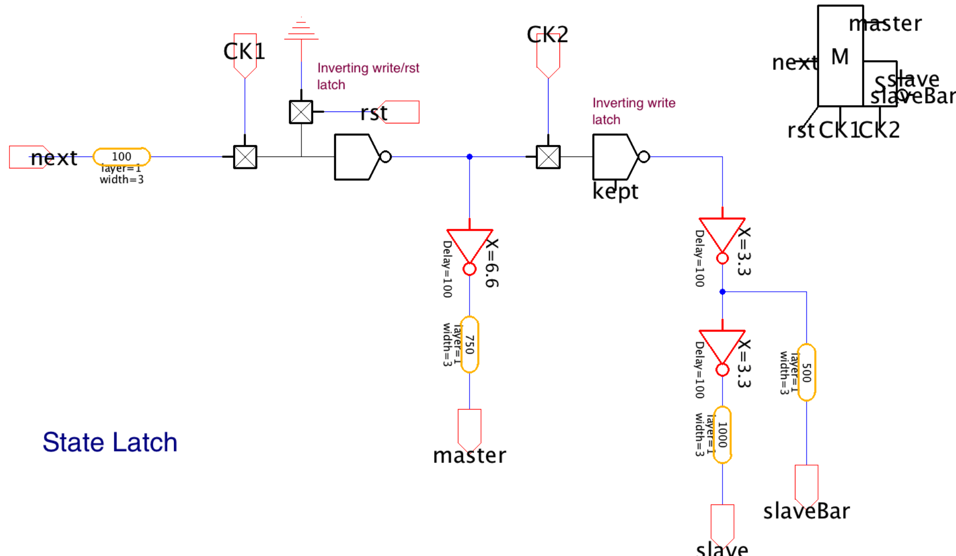


Figure 7: Schematic diagram and icon for the state latch. The internal latches are implemented with complementary pass-transistors and back-to-back non-fighting keepers that operate only when the corresponding latch enable signal (clock) is LO. Two more examples of this latch design style follow in Figure 8 and Figure 9. All our single latches and double-latch master-slave flipflops are designed using this design style. It is stored in Electric_180nm_ARC2013-smg08-09 under jtagCentralSubModules_unchanged as stateBit.

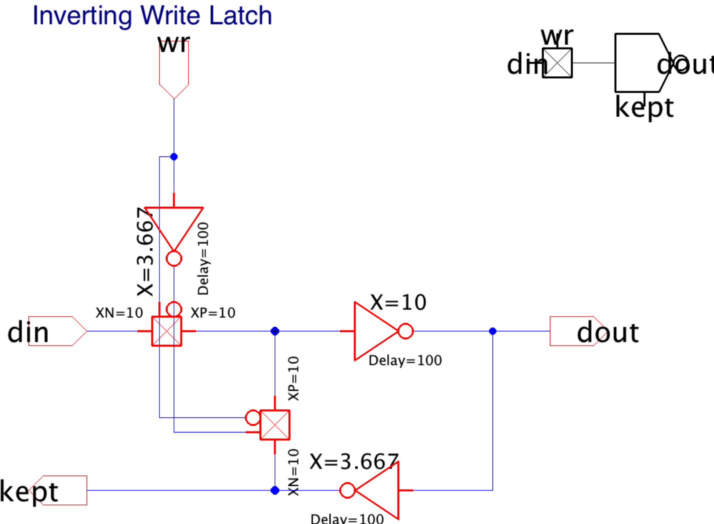


Figure 8: Schematic diagram and icon of a one input one output single inverting latch. Each squared box with a cross denotes a P-N transistor pair that acts as a pass transistor. The two pass transistors in this diagram are driven in mutual exclusion. As a result, there is no need to fight the keeper when writing a new din value to dout. This design is in Electric_180nm_ARC2013-smg08-09, sub-library jtagCentralSubModules_unchanged, where it is stored under the name scan_write.

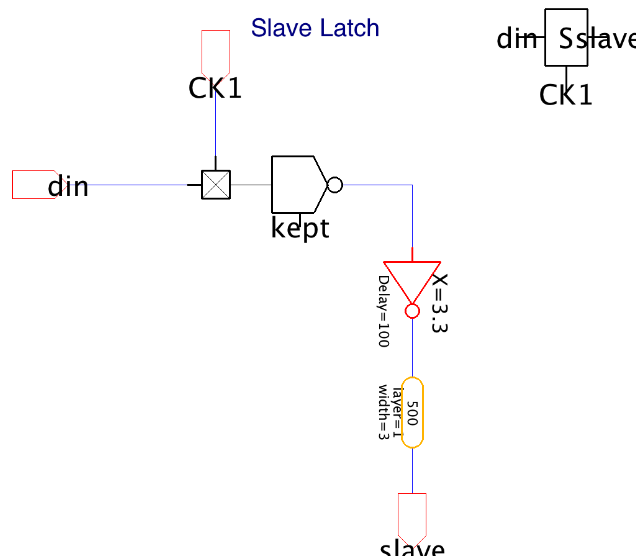


Figure 9: Schematic diagram and icon of a single non-inverting latch, built as extension to Figure 8. This design is stored in Electric_180nm_ARC2013-smg08-09 under jtagCentralSubModules_unchanged as slaveBit.

3.2. Instruction Register (IR) Logic

The JTAG Control Module sets up test instructions that correspond with certain test operations. These test operations can range from built-in self-test and instructions for testing off-chip circuitry to scan chain operations. We will focus mostly on the scan chain operations, for which we added a new instruction mode, called **ScanPath mode**. We use ScanPath mode for scan test operations via the SW/KP and GO scan chains.

The IR Logic has three sub-modules: (1) IR Control, (2) IR, which can operate as shift register, and (3) IR Decode. We'll discuss each sub-module below.

3.1.1 IR Control

Figure 10 shows the design configuration for the IR Control module. As input signals, it uses CK1_in and CK2_in, which are the CK1 and CK2 signals generated in **Figure 2**, and it uses input signals shift, capture, and update, which are TAP Controller output signals ShiftIR, CaptureIR, and UpdateIR in **Figure 2**. From these, the IR Control module generates (1) non-overlapping shift clocks, IRCK1_out and IRCK2_out, that are both LO when we're not shifting IR, and (2) read and write control signals for IR.

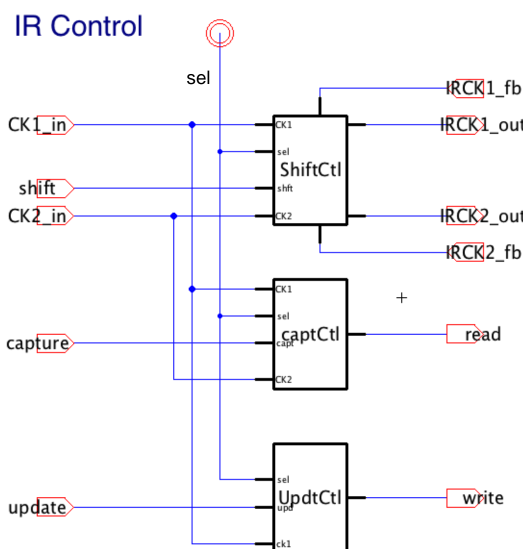


Figure 10: Our design implementation of the IR Control Module in Electric [7]. This design is stored in library Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as jtagIRControl_Scan.

Figure 11 shows the schematic diagram of the IR Shift Control Module, with the internal clock generator to generate non-overlapping IR shift clocks, IRCK1_out and IRCK2_out. To ensure non-overlap HI, the clock generation logic, shown in **Figure 12** also inputs the delayed clock signals, which are fed back through the hierarchy from the farthest out clock distribution points. This is similar to the JTAG Clock Generator module in **Figure 3**. What's different is that both IRCK1_out and IRCK2_out are LO when sel_enable is LO.

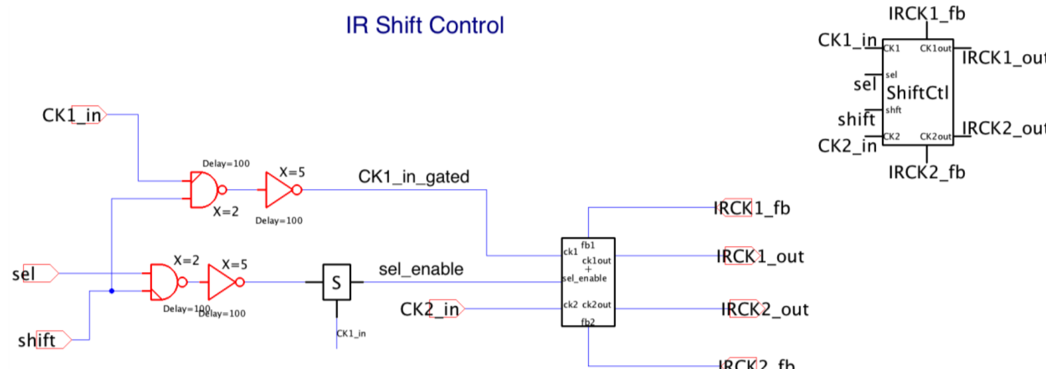


Figure 11: Schematic diagram and icon for the IR Shift Control sub-module. This design is stored in library Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as shift_ctlNew.

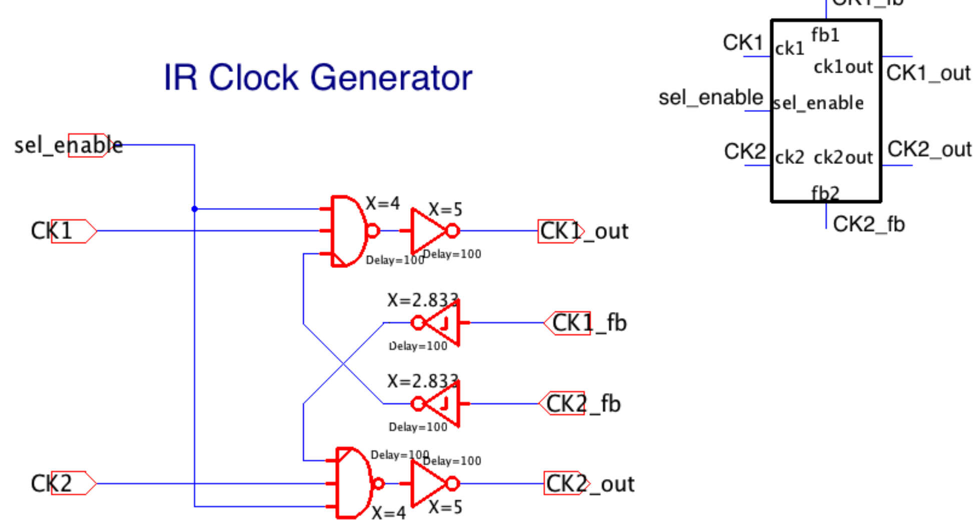


Figure 12: Schematic diagram and icon of the IR Clock Generator module. Input clocks CK1 and CK2 are never HI at the same time. As is the case for the JTAG Clock Generator in **Figure 3**, we can guarantee that CK1_out and CK2_out are never HI at the same time, independent of the delays in the two clock distribution networks. But unlike the JTAG Clock Generator, both CK1_out and CK2_out can be stable and LO at the same time. After stabilizing:

- Either sel_enable or CK1 LO results in CK1_out and CK1_fb LO.
- Either sel_enable or CK2 LO results in CK2_out and CK2_fb LO.
- Both sel_enable and CK1 HI results in CK1_out and CK1_fb HI.
- Both sel_enable and CK2 HI results in CK2_out and CK2_fb HI.

This design is stored in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013, as clockGen_withSel.

From **Figure 11**, we can see that sel_enable is clocked in via a single latch via CK1_in. So, shortly after CK1_in goes HI, sel_enable is set to the logical AND of shift and sel, where sel is HI as it's connected to VDD in **Figure 10**. In other words, when CK1_in goes HI, the value of shift is assigned to signal sel_enable. Signal shift is set in the TAP Controller at CK2, CK2_in here. Having non-overlapping HI CK1_in and CK2_in makes this a proper delay-insensitive assignment. So, signal sel_enable is guaranteed stable when CK2_in changes, thus causing a proper delay-insensitive sel_enable-CK2 input scenario for the IR Clock Generator.

Likewise, signal CK1_in_gated, which is the logical AND of shift and CK1_in, is guaranteed stable and LO when signal shift changes, because CK1_in and CK2_in are never HI at the same time, as is guaranteed by the JTAG Clock Generator in **Figure 3**. When CK1_in rises, signal shift is guaranteed either stable LO or stable HI. For shift LO and stable, CK1_in_gated will remain LO and stable. For shift HI and stable, CK1_in_gated will rise when CK_in rises, and so will sel_enable, Thus, both cases, shift stable LO and shift stable HI, create a proper delay-insensitive sel_enable-CK1 input scenario for the IR Clock Generator.

IRCK1_out may be LO when CK1_in is HI, but when IRCK1_out is HI so is CK1_in. Likewise, when IRCK2_out is HI, then CK2_in is also HI for the IR Control module. We will maintain the phase relation IRCK1_out HI to CK1_in HI to TCK HI, and ditto IRCK2_out HI to CK2_in HI, to TCK LO, throughout our test design.

The incoming capture signal is translated into an IR read control signal, and the incoming update signal is translated into an IR write control signal. Remember that the incoming capture and update signals are outputs from the TAP Controller, so they change at CK2_in.The read and write signals are inputs to the IR, and operate after the IRCK2_out entry point in the slave latch of the shift register. So, to make these read and write control signals fit in our delay-insensitive clocking strategy, they must be changed on IRCK1_out. This is indeed how they're changed in the designs of the sub-modules captCtl and UpdtCtl in **Figure 13** and **Figure 14**. Note that input signal sel for both these sub-modules is tied to VDD in the overall design of the IR Control module in **Figure 10**.

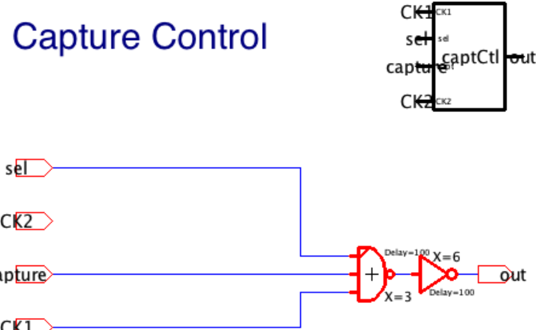


Figure 13: Schematic diagram and icon of the Capture Control (captCtl) sub-module. This design is stored in library Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as capture_ctlNew.

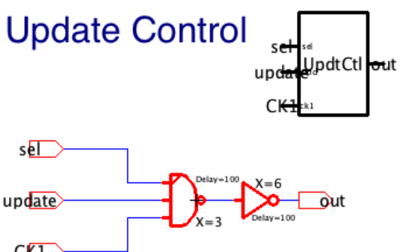


Figure 14: Schematic diagram and icon of the Update Control (UpdtCtl) sub-module. This design is stored in library Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as update_ctlNew.

3.1.2 Instruction Register (IR) – Shift Register Part

Figure 15 shows our IR design with nine serial-in-parallel-out 9-bit shift registers of type scanIRL, with output sout initialized LO, and scanIRH, with output sout initialized HI.

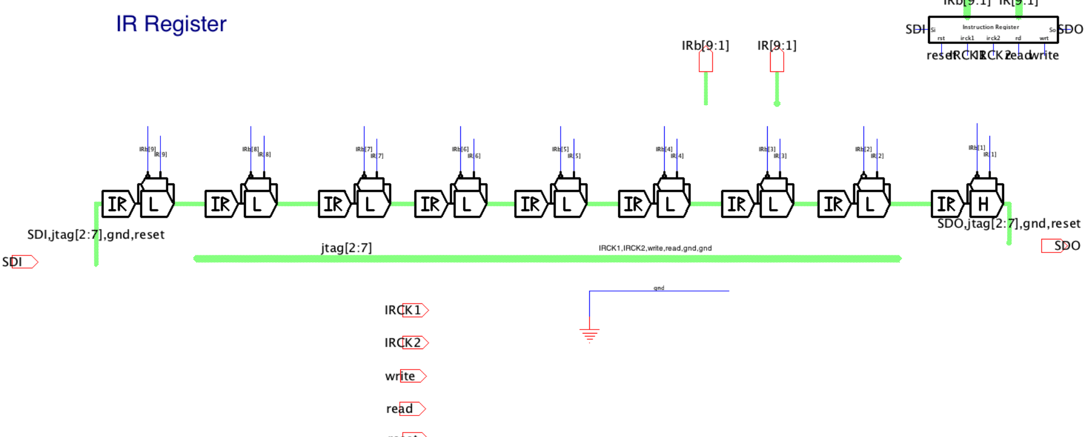


Figure 15: Schematic diagram and icon of the Instruction Register (IR) shift register part. This design is stored in library Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as IR_New.

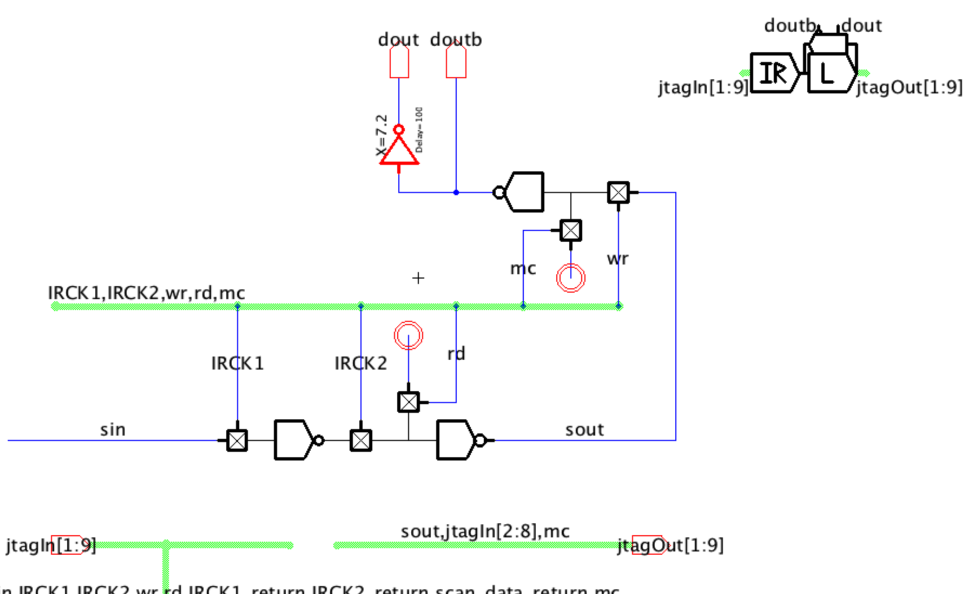


Figure 16: Schematic diagram and icon of the Scan IRL register used in the shift register part of the IR. This design is stored in Electric_180nm_ARC2013-smg08-09, jtagCentralSubModules_unchanged, as scanIRL.

Figure 16 shows the design of the scanIRL register, with its master latch clocked on IRCK1 and its slave latch clocked on IRCK2. It also shows the read (rd) and write (wr) control inputs generated by the IR Control module on the IRCK1 clock, and coming in after the IRCK2 entry point, to maintain a proper delay-insensitive control scheme.

The TAP Controller never overlaps reset, shift, update, and capture, so scanIRL and likewise scanIRH are guaranteed free of drive conflicts between mc (reset), IRCK1 and IRCK2 (shift), wr (update) and rd (capture). With that, scanIRL operates as follows:

- Always dout = ~doutb.
- If IRCK1 goes HI followed by IRCK2 going HI then sout = sin.
- If rd = HI then sout = LO.
- If mc = HI then dout = HI.
- If wr = HI then dout = sout

The full nine-register long shift register in **Figure 15** takes TDI as the input data for SDI and uses the two non-overlapping clocks IRCK1 and IRCK2 to shift the data serially out to SDO which can then be sent via the TDO Driver to test output data TDO. For proper operation, SDI must be stable shortly before IRCK1 rises, which at the level of the JTAG Control module translates to: SDI must be stable shortly before CK1 rises, which at the JTAG IO level translates to: SDI must be stable shortly before TCK rises.

Upon write HI, the shift register contents are copied in parallel to IR[9:1] and its complement IRb[9:1]. Of these, IR[6:1] and IRb[6:1] are connected to the IR Decoder and IR[7:9] are connected to the Output Logic in the JTAG Control module in **Figure 4**.

The nine bits in the IR shift register code the target test instruction that's to be executed next. The operational meaning of each IR bit in the target test instruction is as follows:

- IR[9]: represents the scan clock overlap_enable signal:
  when HI, the scan shift clocks for the selected scan chain will be overlapping HI.
- IR[8]: represents the scan capture signal:
  when HI, the scan capture signal for the selected scan chain will be HI, which will result in reading the corresponding GasP signal value into the scan chain.
- IR[7]: represents the scan update signal:
  when HI, the scan capture signal for the selected scan chain will be HI, which will result in writing the scan values to the corresponding GasP signal.
- IR[6:5]: don't play a role in the scan test experiments in [5].
- IR[4:1]: determines the selection of one out of 13 possible target scan chains, using a 1-hot encoding from 0000 to 1100. The remaining codes 1101 to 1111 are reserved for non-scan test modes and not used in our scan test experiments.

Table 1 summarizes this for quick reference:

IR[9]	IR[8]	IR[7]	IR[6:5]	IR[4:1]
overlap_enable	capture	update	-	0000 to 1100: SelDR[12:0] 1101 to 1111: non-scan test mode

Table 1: Operational intent of each of the 9 bits in the JTAG Instruction Register (IR).

3.1.3 Instruction (IR) Decoder

Figure 17 shows the design of the IR Decoder module. The module takes IR outputs IR[6:1] and IRb[6:1] and generates ScanPath[0:12] and three test mode signals, ByPass, SamplePreload and ExTest. Signal Bypass goes to the Bypass Control module in the JTAG Control module in **Figure 4**, and the other signals go to the Output Logic.

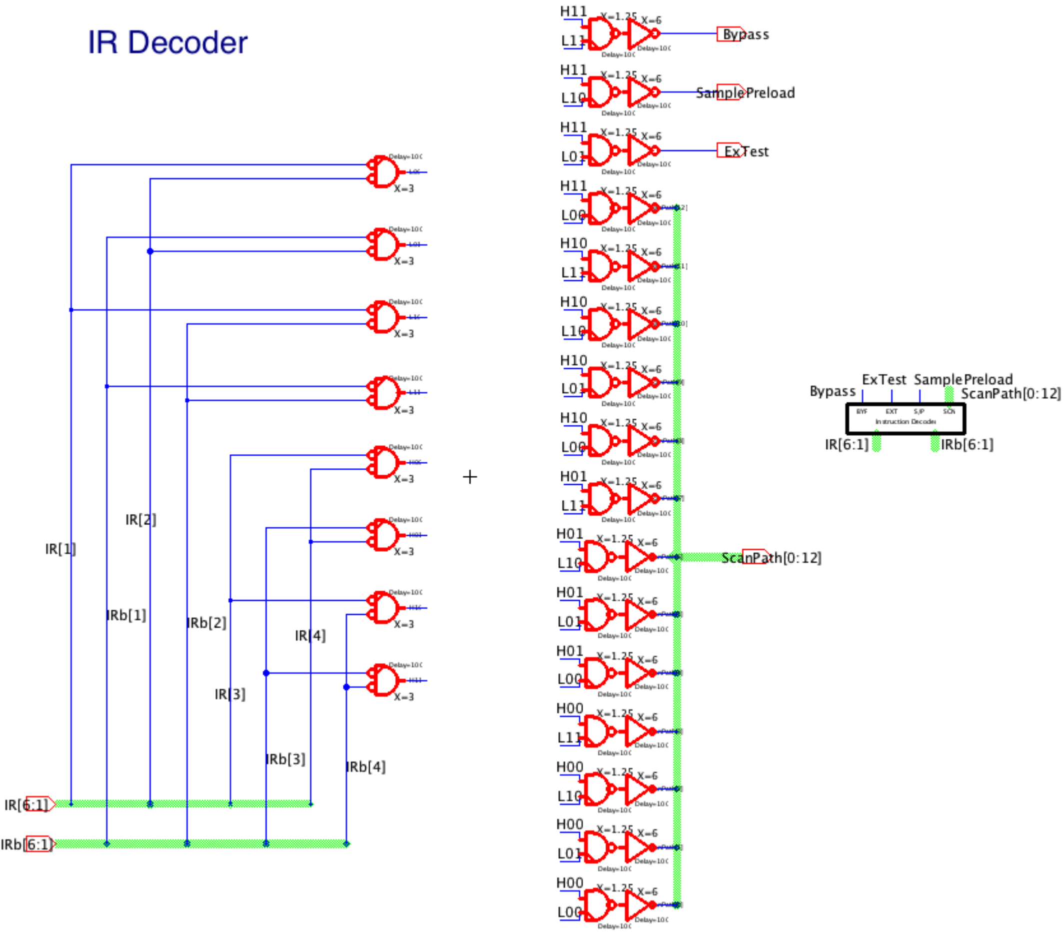


Figure 17: Schematic diagram and icon for the IR Decoder module. This design is stored in directory Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralSubModules_unchanged as IRdecode.

3.3. TDO Driver

The Test Data Output (TDO) Driver multiplexes data from (1) the IR shift register and (2) the currently active scan chain, and outputs the data onto the TDO pin for test inspection. The multiplexed data change on CK2. This matches with the clock ordering in the TDO Driver in **Figure 18**, where the input data is clocked in on CK1 and then propagated as Test Data Output (TDO) on CK2. The second CK2 stage is needed because the IEEE JTAG Controller specification requires that TDO becomes available at the falling TCK clock edge, which corresponds to CK2 rising. See also our discussion about this in relation to the Sun-Oracle design at the end of Section 2.

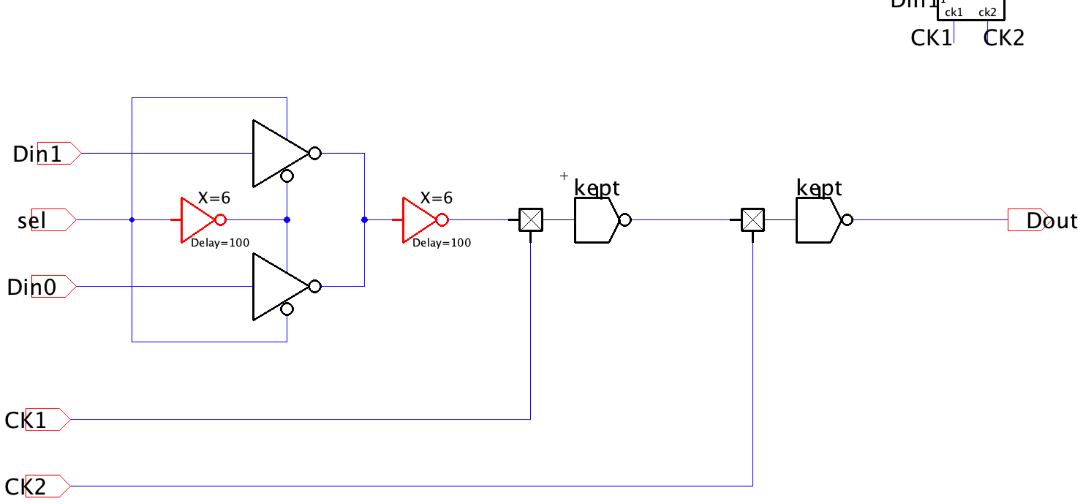


Figure 18: Schematic diagram and icon for the Test Data Output (TDO) Driver module. This design is stored in directory Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as TDOdriver.

3.4. Output Logic

The Output Logic passes the following signals from the JTAG Control to the Scan Control Group in the JTAG Central Module in **Figure 1**:

- SelDR[0:12]: one-hot scan chain selection.
- overlap_enable: control signal to allow overlapping-HI scan shift clocks.
- Capture (Cap): control signal to write the GasP values to the scan chain.
- Update (Upd): control signal to write the scan values to the GasP signals.
- Shift: control signal to shift the scan chain.
- SelBS and ExTest: are unused in the scan chain experiments reported here.
  - SelBS: is used for SAMPLE/PRELOAD JTAG test operations.
  - ExTest: is used for external JTAG test operations.

4. JTAG Central Module

The JTAG Central module combines the JTAG Control module with the Scan Control Group. The Scan Control Group forms the interface to the scan chain, e.g. to the SW/KP and GO scan chains in **Figure 1**. The Scan control group can drive up to thirteen scan chains. It drives them in mutual exclusion, i.e., one at a time.

In our case, each scan chain has eight signals. This is different than the original design by Sun Microsystems, which uses a nine-th so-called master clear signal, mc, to clear all statewires by making them LO or EMPTY. We anticipate that this difference will not matter in our ability to re-use the test software from Sun-Oracle: we just won't connect the mc signal to the design under test, and we'll initialize the circuit via the scan chains.

4.1. Scan Control Group

As we can see in 1 and 2, the Scan Control Group receives its inputs from (1) the Output Logic in the JTAG Control module and (2) the TDI Pulse Setup module in the JTAG Box. It distributes these to thirteen Scan Control modules, one per scan chain. The interface between scan chain and Scan Control module is an 8-bit bus, called leafi[1:8] with i being the index for the module, which has the following signals [5]:

- leaf[1:8] = sin, SCCK1, SCCK2, wr, rd, SCCK1_return, SCCK2_return, sin_return
  - sin: is the scan-data input to the scan chain.
  - SCCK1, SCCK2: are the scan clock signals for shifting data in and out.
  - rd: is the clock-enable signal for readin data from the design into the scan chain.
  - wr: is the clock-enable signal for writing data from the chain into the design.
  - SCCK1_return, SCCK2_return: are the far-out signals for SCCK1 and SCCK2.
  - sin_return: is the scan-data output of the scan chain.

Figure 19 and **Figure 20** show the Scan Control Group and Scan Control Module designs. Lower level details of the Scan Control Module follow in **Figure 21** to **Figure 23**.

The input-output relation of the Scan Control module in **Figure 20** is as follows:

- JTAG inputs **overlap_enable** and **shift** are combined with the incoming module selector **sel[i]** and the two non-overlapping-HI incoming clocks **CK1** and **CK2** and parsed by sub-module Scan Shift Control (ShiftCtl) to generate the two output scan shift clocks **SCCK1 (leaf[2])** and **SCCK2 (leaf[3])** and their far-out wire connections **SCCK1_fb (leaf[6])** and **SCCK2_fb (leaf[7])**. The far-out clock signals are used by sub-module ShiftCtl to ensure that the generated clocks are non-overlapping HI when needed, by using design techniques similar to those we used in the designs of the JTAG and IR Clock Generators in **Figure 3** and **Figure 12**. The design of sub-module ShiftCtl follows in **Figure 22**.
- Input **cap** for capture is combined with the incoming module selector **sel[i]** and non-overlapping-HI incoming clocks **CK1** and **CK2** and parsed by sub-module Scan Capture Control (captCtl) to generate output **rd (leaf[5])** which reads the values of the design signals into the scan chain. Scan Capture Control is an instance of the same design as IR Capture Control, shown earlier in **Figure 13**.

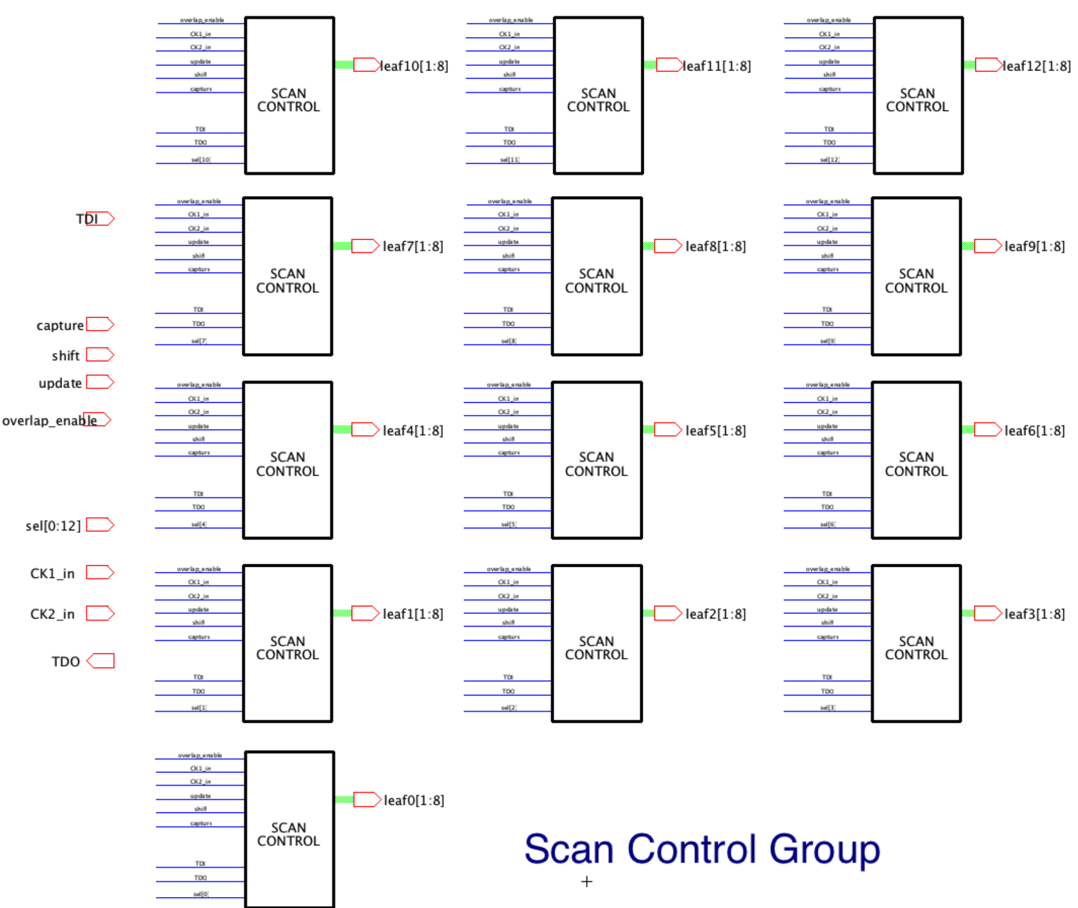


Figure 19: Schematic diagram the Scan Control Group, showing all thirteen Scan Control Modules, one for each scan chain. The input ports on the left are connected to all thirteen modules. The mutually exclusively driven and tri-state outputs of the thirteen modules are connected to the single TDO output of the Scan Control Group. This design is in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013, under jtagScanCtlGroupNew.

- Input **upd** for update is combined with incoming module selector **sel[i]** and incoming clock **CK1** and parsed by sub-module Scan Update Control (UpdtCtl) to generate output **write (leaf[4])** which writes the scan values into the design. Scan Update Control is an instance of the same design as IR Update Control, shown earlier in **Figure 14**.
- Input **TDI** is amplified and output as **leaf[1]**, which acts as scan data input.
- Scan bus signal **leaf[8]** is the output data of the scan chain. It is combined with the incoming module selector **sel[i]** and parsed by sub-module Scan Control which, if sel[i] is HI, passes **leaf[8]** to TDO, and which otherwise tristates its connection to TDO.

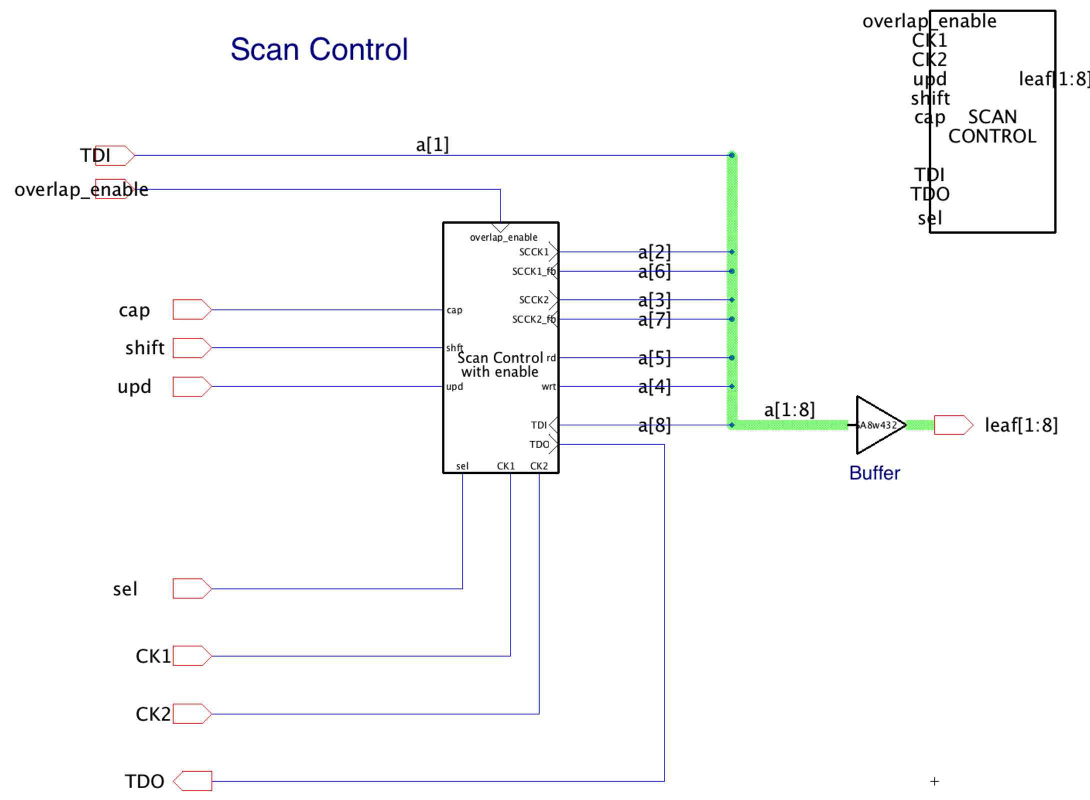


Figure 20: Schematic diagram and icon of a Scan Control Module. This design is stored in directory Electric_180nm_ARC2013-smg08-09 under sub-library jtagCentralLatest2013 as jtagScanCtlWBufNew.

Note that the organization details of sub-module "Scan Control with enable" in **Figure 21** are similar to those of the IR Control module in **Figure 10**. Like the IR Control module, module "Scan Control with enable" has three key modules: (1) ShiftCtl to generate the scan shift clocks, (2) capCtl to translate the capture signal into a scan read control signal, and (3) updtCtl to translate the update signal into a scan write control signal.

Remember that the incoming capture and update signals are outputs from the TAP Controller, so they change at CK2. The read and write signals are inputs to the scan chains, and operate after the SCCK2 entry point in the slave latch of the scan shift register. So, to make these read and write control signals fit in our delay-insensitive clocking strategy, we change them while CK1 is HI (and thus SCCK2 is LO). Note that this is indeed how they're changed given the connectivity in **Figure 19** to **Figure 21** and the designs of the two modules captCtl and UpdtCtl in **Figure 13** and **Figure 14**.

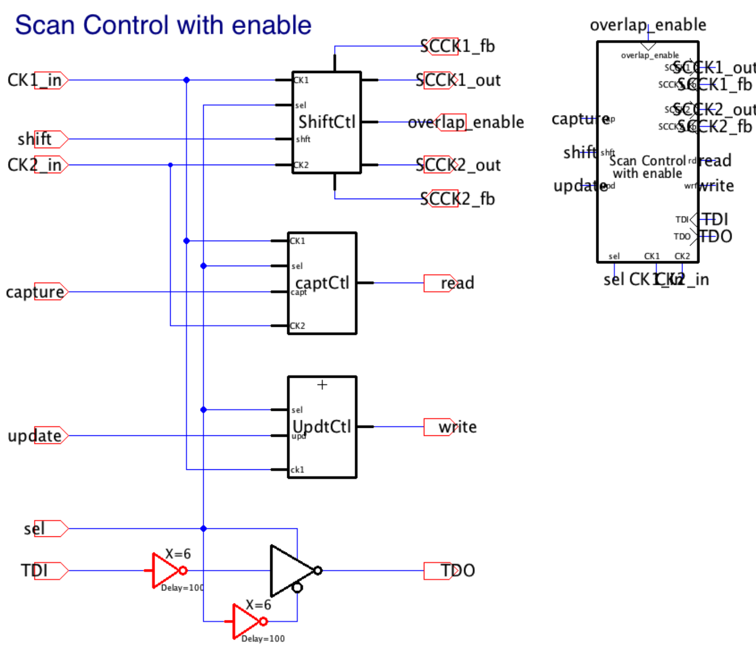


Figure 21: Schematic diagram and icon of sub-module Scan Control with Enable. Notice the similarity in organization with module IR Control in Figure 10. Sub-modules captCtl and UpdtCtl are instances of the same designs as those for module IR Control. The design instance for submodule ShiftCtl is different because it makes both outgoing clocks SCCK1_out and SCCK2_out HI when both sel and overlap_enable are HI. In contrast, the ShiftCTL submodule in module IR Control lacks the overlap_enable input and always generates clocks with non-overlapping HI pulses. Input pin TDI is the TDO output pin of the scan chain associated with this controller. The TDI to TDO pin connection to the TDO pin of the overall Scan Control Group is established when the scan chain is selected, i.e., when sel is HI, and otherwise, i.e., when sel is LO, it's tri-stated. This design is in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013, under jtagScanControl_mcNew.

The design differences between the sub-module "Scan Control with enable" in **Figure 21** and the IR Control module in **Figure 10** are (1) an additional tri-state TDI to TDO connection in the scan controller, where TDI is the local TDO output pin of the scan chain associated with the controller, and (2) an additional clock generation mode in the shift control block, ShiftCtl, that makes both generated clocks HI whenever the scan control inputs overlap_enable and sel are HI; otherwise clock generation is the same.

The designs for the Scan Shift Control block and the Scan Control Clock Generator follow in **Figure 22** and **Figure 23** below.

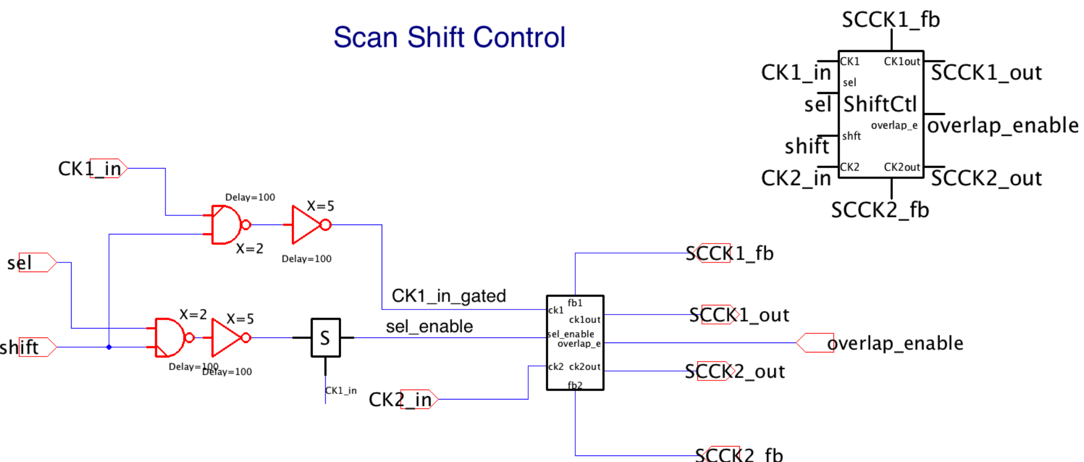


Figure 22: Schematic diagram and icon for the Scan Shift Control module. The difference with respect to the earlier IR Shift Control module shown in Figure 11 is in the Clock Generator, whose design follows in Figure 23. This design is in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013, under shift_ctl_mcNew.

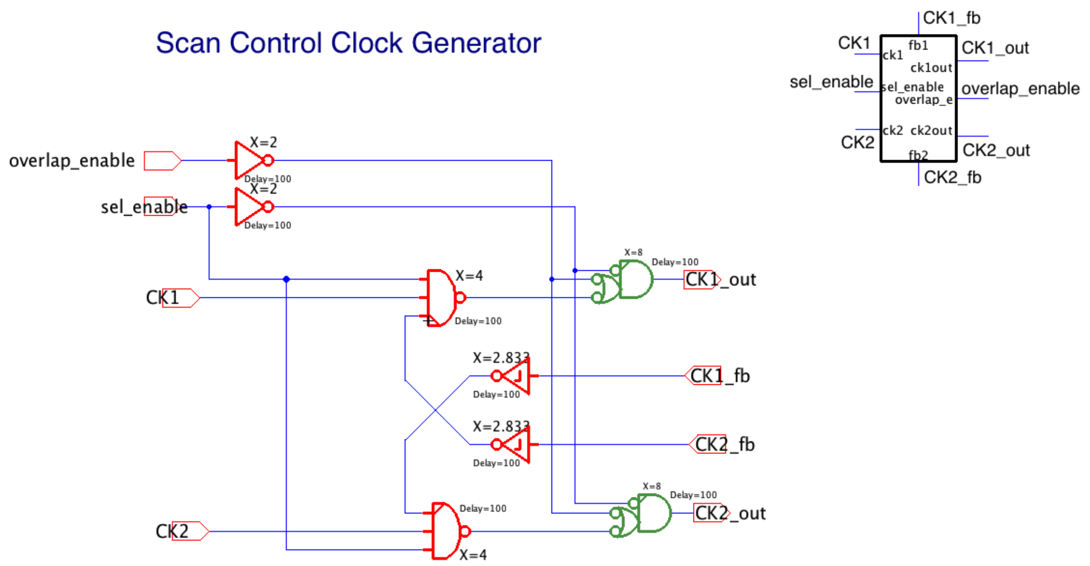


Figure 23: Schematic diagram and icon of the Scan Control Clock Generator. In addition to the functionality of the IR Control Clock Generator module presented in Figure 12, the above diagram pulls both CK1_out and CK2_out and HI whenever overlap_enable is HI and the associated scan chain is selected, i.e., sel_enable is HI. After stabilizing, the above diagram behaves as follows:

- Either sel_enable or CK1 LO results in CK1_out and CK1_fb LO.
- Either sel_enable or CK2 LO results in CK2_out and CK2_fb LO.
- Both sel_enable and CK1 HI results in CK1_out and CK1_fb HI.
- Both sel_enable and CK2 HI results in CK2_out and CK2_fb HI.
- Both sel_enable and overlap_enable HI results in CK1_out, CK1_fb, CK2_out, and CK2_fb HI.

This design can be found in directory Electric_180nm_ARC2013-smg08-09, sub-library jtagCentralLatest2013, under clockGen_withSelandOverlap.

5. JTAG Box Programming Interface

To ease the design of our scan tests in Electric [7], we use a programming interface, which we call “Pulse Setup”. The Pulse Setup interface provides a more abstract and user-friendly interface to program signal pulses for (1) JTAG Central module inputs TDI, TCK, TMS, TRSTb and (2) scan chain input TDI. We typically generate the TCK as a continuous clock signal, which starts clocking after a given offset time. For TDI, TMS, and TRSTb, we generate non-continuous pulses by using Piece Wise Linear (PWL) components from our Electric design library. A piece-wise linear specification indicates the corner points, i.e., points of change, in the (Time x Voltage) pulse function; the changes in-between these points are linear in the Time-Voltage domain. **Figure 24**(right) is an example of a piece-wise linear specification. We use a 180nm process technology, where LO corresponds to 0 Volt and HI to 1.8 Volt.

Our test sequences starts with TCK and TRSTb LO, to unambiguously reset the TAP Controller. After some time, we set TRSTb to HI, which ends the reset phase, start the continuous clock function of TCK, and then start toggling TMS as needed to traverse the TAP controller to the desired states from where we toggle TDI to program the desired test instruction or the desired scan values, whichever one the given TAP Controller state allows us to program. Details follow in Sections 5.1 to 5.3 below.

5.1. TCK and TRSTb Pulse Setup

For TCK, we use a clock period of 5 ns, a HI pulse width of 2.3 ns, 200 ps rise and fall times, and a start time of 23.5 ns. The resulting LO pulse width is 2.3 ns. We start with TRSTb LO (0 Volt) for 20 ns, and make it HI (1.8 Volt) 0.2 ns later and for the rest of the (under 20 seconds) simulation. This ensures that the TAP Controller is reset when TCK starts its clock function at 23.5 ns. The pulse components and an initial segment of their waveforms follow in **Figure 24** and **Figure 25** below.

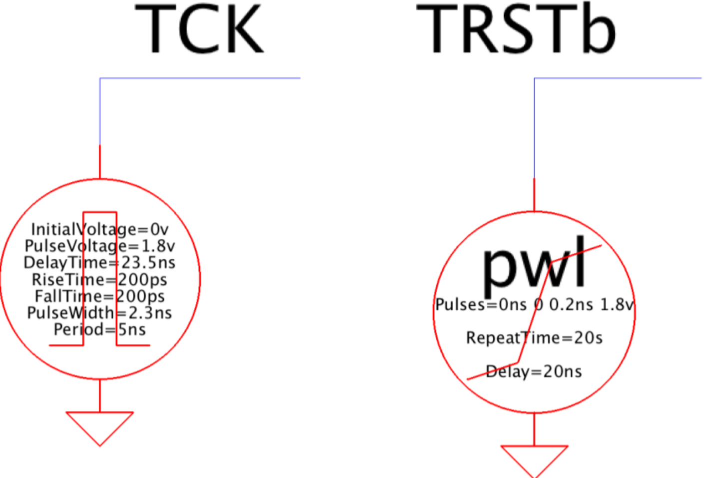


Figure 24: Pulse components for JTAG Control inputs TCK and TRSTb. These designs are stored in directory Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013 as PULSE_generator and PWL_generator.

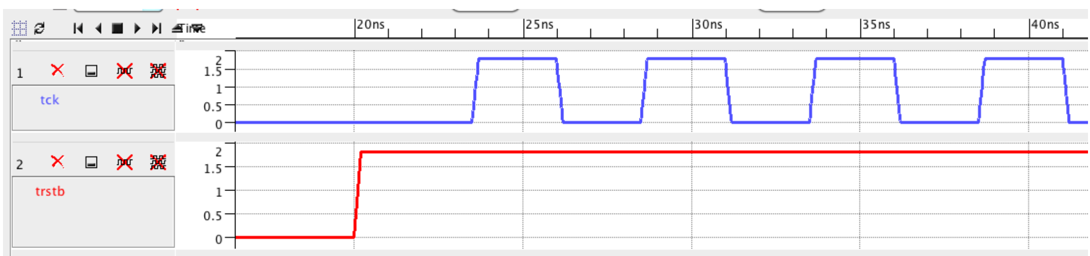


Figure 25: Initial segment of the pulse waveforms for the TCK and TRSTb pulse components of Figure 24.

5.2. TMS Pulse Setup

The strategy is to first load an instruction in the IR Logic and then update the scan data. Consequently, the TAP controller traverses first the IR control states and then the DR control states. **Figure 5** shows that this leads to the following 4-step algorithm:

1. Set the TAP Controller in the IDLE state for a duration of zero or more cycles. During initialization, this happens automatically provided we keep TMS LO while the TAP Controller is reset and for the first few TCK clock cycles after reset, as explained in Section 5.1.
2. Set-IR: Go via SELECT_DR_SCAN to SELECT_IR_SCAN to CAPTURE_IR, on to SHIFT_IR for as many TCK cycles as needed to shift in the next instruction, and then via EXIT1_IR to UPDATE_IR to hand over the new instruction.
3. Go to IDLE for a positive number of TCK cycles, or skip IDLE (default).
4. Set-DR: Go via SELECT_DR_SCAN to CAPTURE_DR, on to SHIFT_DR for as many TCK cycles as needed to shift in the next scan values, and then via EXIT1_DR to UPDATE_DR to hand over the scan data.

Figure 26 shows the pulse component for a full algorithmic cycle from step 1 to step 1, assuming 9 TCK clock cycles to shift in a full 9-bit instruction during SHIFT_IR, and assuming 5 TCK clock cycles to shift in five GO values for the 5-stage FIFO of **Figure 1**.

From Section 3.1, we know that TMS must be set up prior to TCK going LO, and before TCK goes LO again. The Delay value for the TMS pulse generator must be set such that TMS changes while TCK is HI and sufficiently before TCK goes LO. In this report, we have used TMS Delay=40ns, i.e.,:

- TMS Delay = TCK Delay (23.5 ns) + 3 TCK HI-LO cycles (15 ns) + 1.5 ns in TCK HI,
- which implies that the TMS Delay ends while TCK is HI and 1 ns before TCK falls,
- which is good enough for the simulations for the small-scale design in this report.

This also implies that TMS is LO for the first 3 TCK clock cycles, which guarantees that the TAP Controller has had the opportunity to transition from the RESET to the IDLE state, before we start the TMS pulse sequence.

The supporting waveforms, relating TMS to TCK, CK1, CK2, and the RESET to IDLE transition in the Tap Controller follow in **Figure 27** below.

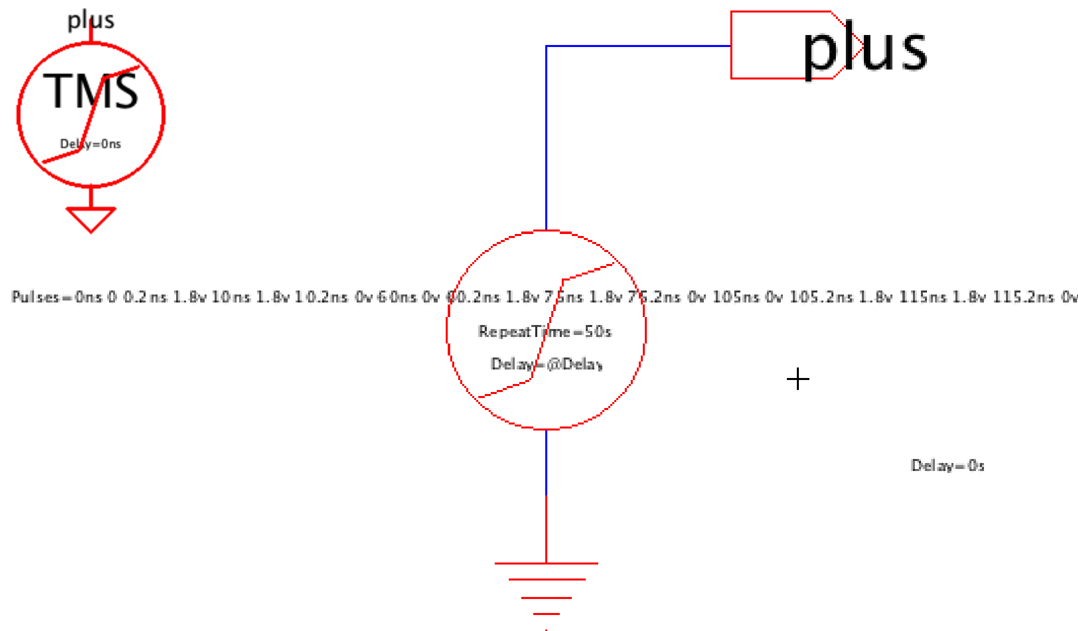


Figure 26: Pulse component and icon (top-left) for JTAG Control input TMS. This TMS pulse generator loads a full 9-bit instruction and five scan values, which we'll use to scan in either 5 GO signals or 4 SW values. The default TMS Delay value is 0 ns, as shown here. For the simulations in this ARC report, the TMS Delay is 40 ns. This design is in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013,as IR_DR_tms_go_generator.

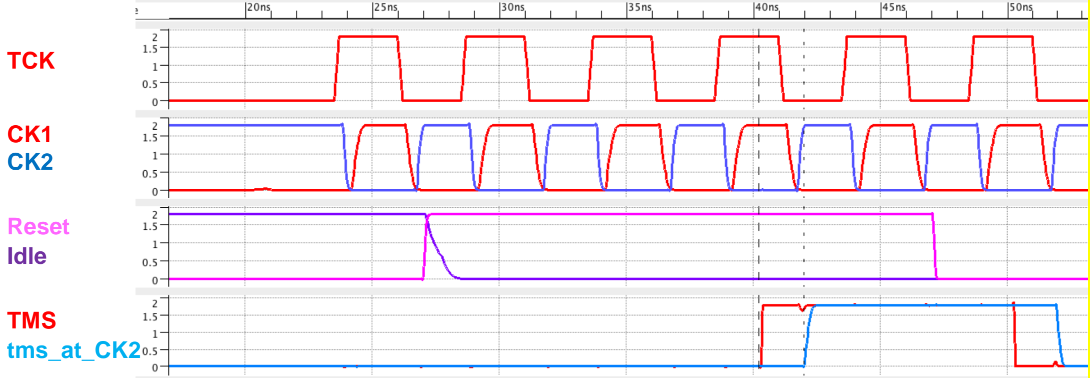


Figure 27: Relation between the waveforms (1) generated by the pulse components for TCK in **Figure 24**(left), and (2) the TCK resulting CK1 and CK2 waveforms generated by the JTAG Clock Generator in **Figure 3**, and (3) the RESET and IDLE state transitions in the Tap Controller, and (4) the TMS pulse component in **Figure 26**, where TMS is the signal generated by the pulse component and tms_at_CK is the version that's distributed during CK2 HI in our implementation of the TAP Controller in **Figure 6**. The gap between the two vertical dashed lines around 40ns shows that we have an adequate setup time for TMS with respect to CK2 rising.

The TMS pulse generated by the pulse component in **Figure 26** translates into the following bit vector:

- 110[0^9]1110[0^5]110
- where:
  - 0 means LO and 1 means HI,
  - each bit represents the TMS value seen during the HI TCK clock pulse, and counted from the point in time when the parameterized Delay expires.
  - bit values in red indicate TMS values input during the SHIFT_IR state, and those in blue indicate TMS values input during the SHIFT_DR state.

Ideally the TMS pulse component itself could have provided a bit level user interface, with setup and hold time settings for the bits in relation to TCK, but this is not possible in the underlying SPICE environment. Instead, the valid setup and hold time settings and the valid bit values must be programmed manually by carefully programming both the piece-wise linear sequence of time-voltage pairs and the Delay value so that in relation to the TCK parameters the bits line up adequately with each TCK HI pulse.

We expect that a Verilog-supported component would be able to directly provide a more abstract and scalable bit-level user interface.

The relation between the TMS values and the TAP Controller state transitions through the Finite State Machine configuration in **Figure 5** is illustrated more clearly in **Table 2**, and is supported by the SPICE simulated waveforms in **Figure 28**.

	INIT	**Set-IR**			**Set-DR**			**IDLE**
TAP States	RESET IDLE	SELECT_DR_SCAN SELECT_IR_SCAN CAPTURE_IR	SHIFT_IR	EXIT1_IR UPDATE_IR	SELECT_DR_SCAN CAPTURE_DR	SHIFT_DR	EXIT1_DR UPDATE_D R	IDLE
TMS pulse	[0]	110	00000000 0	11	10	00000	11	0

Table 2: The stages of the TMS Pulse Setup algorithm (top row) set against the TAP controller states traversed through the Finite State Machine in **Figure 5** (middle row) and the corresponding TMS values (lower row).

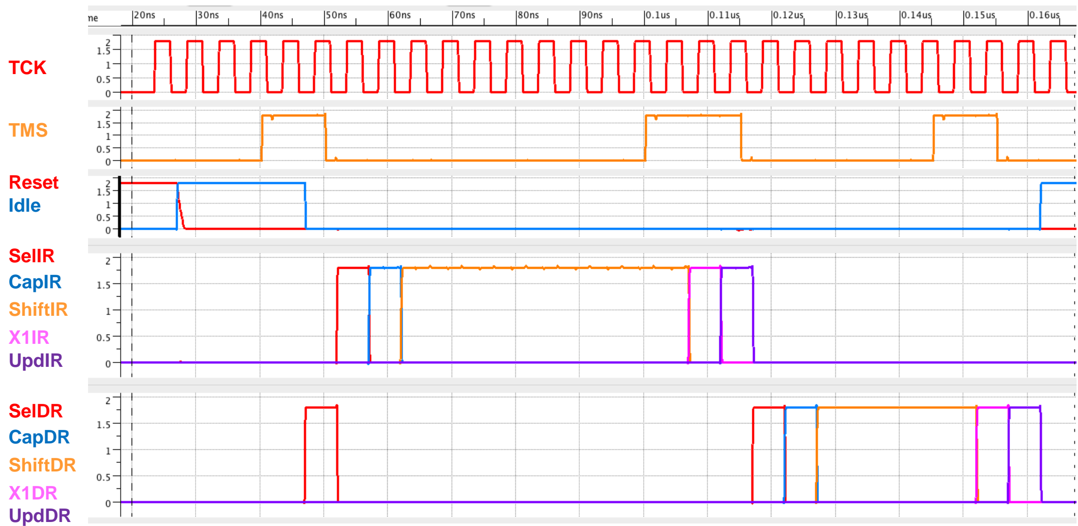


Figure 28: Relation between the TMS values generated by the TMS pulse component in **Figure 26** and the states traversed in the Tap Controller at each TCK HI-LO cycle. Note that there are exactly 9 HI-LO TCK clock cycles when ShiftIR is high, i.e., in the SHIFT_IR state, and exactly 5 HI-LO TCK cycles when ShiftDR is high.

5.3. TDI Pulse Setup

TDI is an input for (1) the shift register in the IR Logic sub-module in the JTAG Control module and (2) the Scan Control Group module which directs it to the scan chains. The modules use TDI during different periods of the test execution. The IR data values are shifted in during the SHIFT_IR state of the TAP Controller, i.e., at the red TMS LO bits in **Table 2**, and the scan data values are shifted in during the SHIFT_DR state, i.e., upon the blue TMS LO bits in **Table 2**. They are shifted in at the rising edge of TCK.

From **Table 2**, we can see that a total of nine (9) TDI bits are shifted into the IR shift register part during the SHIFT_IR state of the TAP Controller, and that the first and last of these TDI bits are shifted in during the 4-th and 12-th TMS bit, respectively.

More precisely, based on the design of the state latches (**Figure 7**) used in the TAP Controller (**Figure 6**): TAP Control output ShiftIR goes HI on the 4-th TMS bit at the fall of TCK. So, we can safely delay the TDI component action by setting:

- TDI Delay = TMS Delay + 4.5 TCK cycles = 40 ns + 22.5 ns = 62.5 ns,
- which – by symmetry in the HI-LO-rise-fall TCK pulse generation – ends 1 ns before TCK goes HI, which suffices for the simulations in this report.

Figure 29 shows that the TMS setup time to JTAG Clock Generated output CK2 HI is fine, and so are the TDI setup to IR Clock Generated CK1 HI and the TDI setup time to Scan Control Clock generated SCCK1 HI.

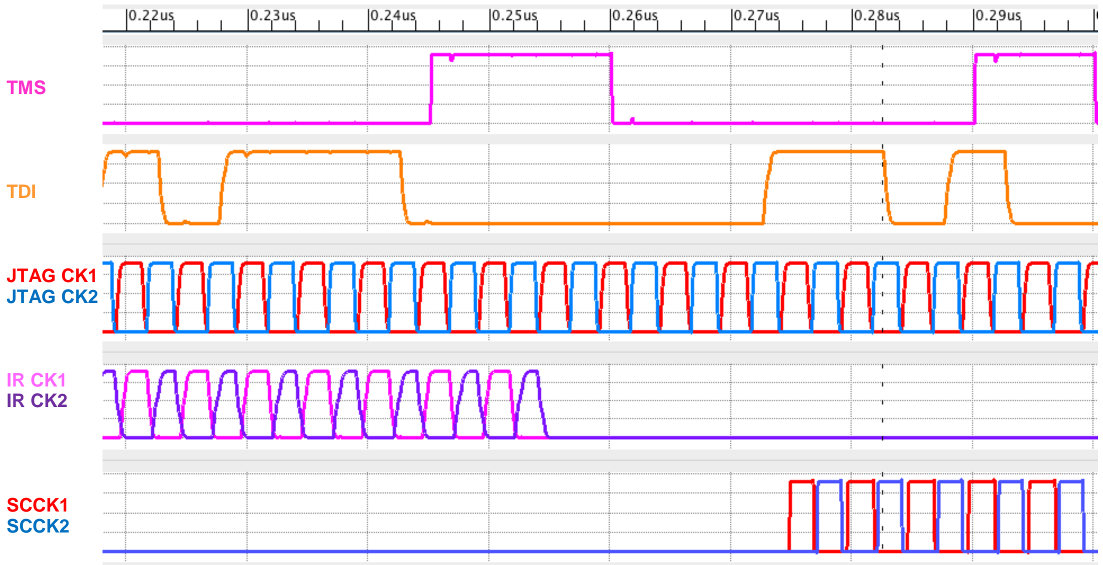


Figure 29: SPICE waveforms for the TMS-TDI Pulse Generators in our scan test operations, showing that the TMI set up time to JTAG Clock Generator output signal CK2 HI is fine. Likewise, the TDI setup time to both the IR Clock Generator output CK1 HI and Scan Control Clock Generator output CK1 HI are fine.

From the IR register design in **Figure 15**, we can see that the shift register bits, as encountered from the TDI side, are ordered from high to low, so the first TDI bit shifted in ends up at IR[1] and the last and nine-th bit shifted in ends up in IR[9].

For example, to shift in instruction IR[9:1]=101110000 for clearing the GO signals when the GO scan chain is attached to leaf0 of the Scan Control Group, we serially shift in TDI bits 000011101 with the first bit, 0, read in on the first TCK cycle after TDI Delay, and the nine-th bit, 1, read on the nine-th TCK cycle after TDI Delay expires. The resulting IR status is shown in Table 2.

IR[9]	IR[8]	IR[7]	IR[6:5]	IR[4:1]
1	0	1	11	0000

Table 3: Bit values in the IR shift register part after serially shifting in the TDI bit values 000011101 on TCK HI. This instruction reads as: overlap the scan clocks (IR[9]), don't read the values of the design signals (~IR[8]) but rather write the scan values into the corresponding design signals (IR[7]), and use the scan chain connected to leaf0 [IR[4:1]. IR[6:5] play no role here. The IR codes are explained in Section 3.1.2 and **Table 1**.

Shifting in five scan bits is similar, except that the first TDI bit shifted in during SHIFT_DR will end up at scan register GO_SCAN[5] for GO[5] and the last bit shifted in will end up at the scan register GO_SCAN[1] for GO[1]. For example, to shift in GO_SCAN[5:1]=11101 we serially shift in TDI bits 11101 with the first bit, 1,shifted in on the first TCK cycle in state SHIFT_DR, and the second to last bit, 0, on the second to last TCK cycle in state SHIFT_DR. The resulting GO_SCAN values follow in
Table 4.

GO_SCAN[1]	GO_SCAN[2]	GO_SCAN[3]	GO_SCAN[4]	GO_SCAN[5]
1	0	1	1	1

Table 4: Bit values in the GO Scan Chain after serially shifting in the TDI bit values 11101 on TCK HI.

From **Table 2**, we can see that the first and last of these TDI bits are shifted in during the 17-th and 21-st TMS bit, respectively. More precisely, based on the design of the state latches (**Figure 7**) used in the TAP Controller (**Figure 6**): TAP Control output ShiftDR goes HI on the 17-th TMS bit at the fall of TCK. So, after shifting in the last TDI bit value for SHIFT_IR, we can safely wait with providing new TDI bits for a duration of:

- TDI skip = 5 TCK cycles = 25 ns.

In-between the TAP Controller states SHIFT-IR and SHIFT-DR, TDI plays no role, as neither (1) the shift register in the IR Logic sub-module in the JTAG Control module nor (2) the Scan Control Group module which directs it to the scan chains are using TDI. The shift register part of IR is then not using TDI because shiftDR is reset to LO upon exiting Tap Controller state Shift_IR at TCK LO, which disables CK1 and CK2 clock propagation in the IR Clock Generator, leaving both shift register clocks disabled at LO – see **Figure 11** and **Figure 12** for details. The same reasoning applies to the Scan Control Clock Generator, as can be seen from **Figure 22** and **Figure 23**.

Figure 30 shows the piece-wise linear pulse component for the TDI signal. It has nine parameters, V1 to V9, to hand over the nine IR shift register values, and five, D1 to D5, to hand over the five scan shift values to either the GO or the SW-KP scan chains. In case a parameter is left unspecified, its voltage level defaults to 1, i.e., 1.8 Volt.

Given the tight TMS-TDI connection, we will combine **Figure 26** and **Figure 30** into one single Pulse Generator, shown in **Figure 31**, for ease of use in our test program setups.

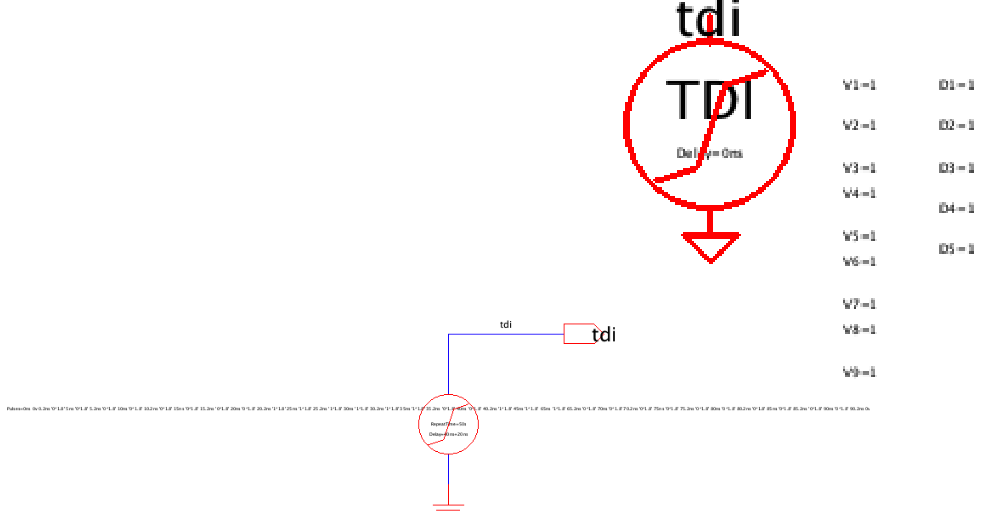


Figure 30: Piece-wise linear (PWL) component and icon (top-right) for the JTAG Control input TDI used in this report. The tiny – and here illegible – sequence of pairs of time(ns)-voltage(V) points for the PWL specification reads as follows:

- **Sequence offset time:**
  TDI Delay = 0 ns by default, but for the purpose of this report, we'll set TDI Delay = 62.5 ns
- **TDI values for IR:**
  (0-0) (0.2-V1) (5-V1) (5.2-V2) (10-V2) (10.2-V3) (15-V3) (15.2-V4) (20-V4)
  (20.2-V5) (25-V5) (25.2-V6) (30-V6) (30.2-V7) (35-V7) (35.2-V8) (40-V8) (40.2-V9)
- **TDI values for DR:**
  (65-V9) (65.2-D1) (70-D1) (70.2-D2) (75-D2) (75.2-D3) (80-D3) (80.2-D4) (90-D4) (90.2-D5)

The default values for V1 to V9 and D1 to D5 are 1, i.e., 1.8 Volt, as indicated in the icon and PWL sequence. This design is in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013, stored as TDI_go_generator.

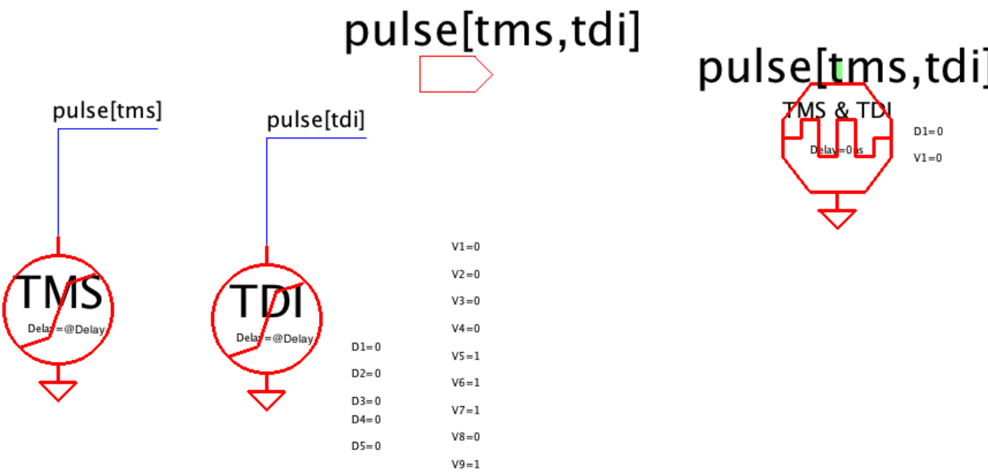


Figure 31: Single TMS-TDI Pulse Generator, with its icon (top-right), combining the TMS and TDI Delays and the shift parameters for the test instruction (V1 to V9) and scan values (D1 to D5) for one algorithmic test cycle. This design is in Electric_180nm_ARC2013-smg08-09, jtagCentralLatest2013, as TMS_TDI_go_generator.

6. Scan Test Experiments: Simulation Setup and Results

Figure 32 shows the schematic diagram of our scan test experiment. It contains the JTAG Box setup with the Pulse Setup and JTAG Central module on the left, and the two SW-KP and GO scan chains and the 5-stage GasP FIFO on the right. Design and SPICE simulations for this test setup were done in 180nm CMOS.

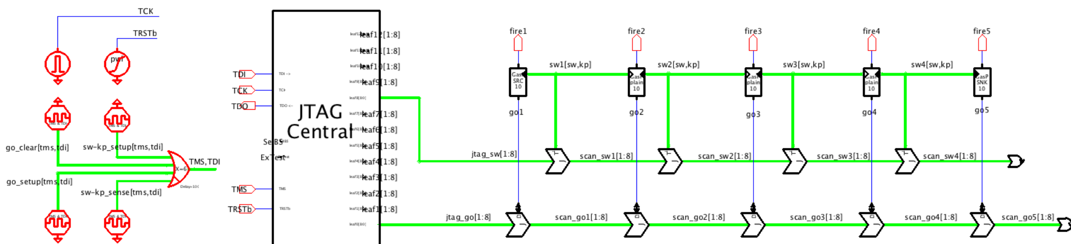


Figure 32: Schematic diagram of the scan test experiments performed in this report. Scan chains and FIFO are organized and ordered from left to right as SW-KP_scanchain[1:4], GO_scanchain[1:5], and FIFO_stages [1:5]. The design and SPICE simulations for this test setup were done in a 180nm CMOS manufacturing process. This design is stored in library JTAG_experiments2013 under Experiment_JTAG_firstN_eight.

The test includes both test experiments that control as well as test experiments that observe the state of the GasP FIFO. We performed the following scan operations:

1. **go_clear [tms,tdi]** set all GO signals to LO.
2. **sw-kp_setup [tms,tdi]** set the statewires to specific values of our choice.
3. **go_setup [tms,tdi]** un-block some GO signals, allowing limited self-timing.
4. **sw-kp_sense [tms,tdi]** read the statewires (after the self-timed operation).

The details of the Pulse Setup program follow in Section 6.1 below, and the test functionality is explained in Section 6.2, along with the SPICE simulation waveforms.

6.1. Programming the Input Pulse Generators

The Pulse Generators for TCK and TRSTb in **Figure 32** are exact replica of those in **Figure 24**. The Pulse Generators for the TMS and TDI pulses are shown in **Figure 33**: there are four separate instances of the combined TMS-TDI Pulse Components à la **Figure 31**, one per scan operation. The scan operations are performed in sequence and after the test circuitry has initialized, which is reflected in the offset Delays.

Each scan operation takes 24 TCK cycles, or 120 ns. So, if we offset the Delays for TCK and TRSTb and TMS[1], as we did before, then all we need to do next to maintain a correct skew relation between test clocks and test data is to delay each subsequent scan operation by an extra (24+n) TCK cycles, for some positive number n. A summary of the overall Delay settings and V1-V9 (IR) parameters and D1-D5 (DR) parameters for this scan test experiment are summarized in **Table 5** below.

[1] We maintain a fixed offset of 22.5 ns from TMS Delay to TDI Delay, as we did in Section 5.3.

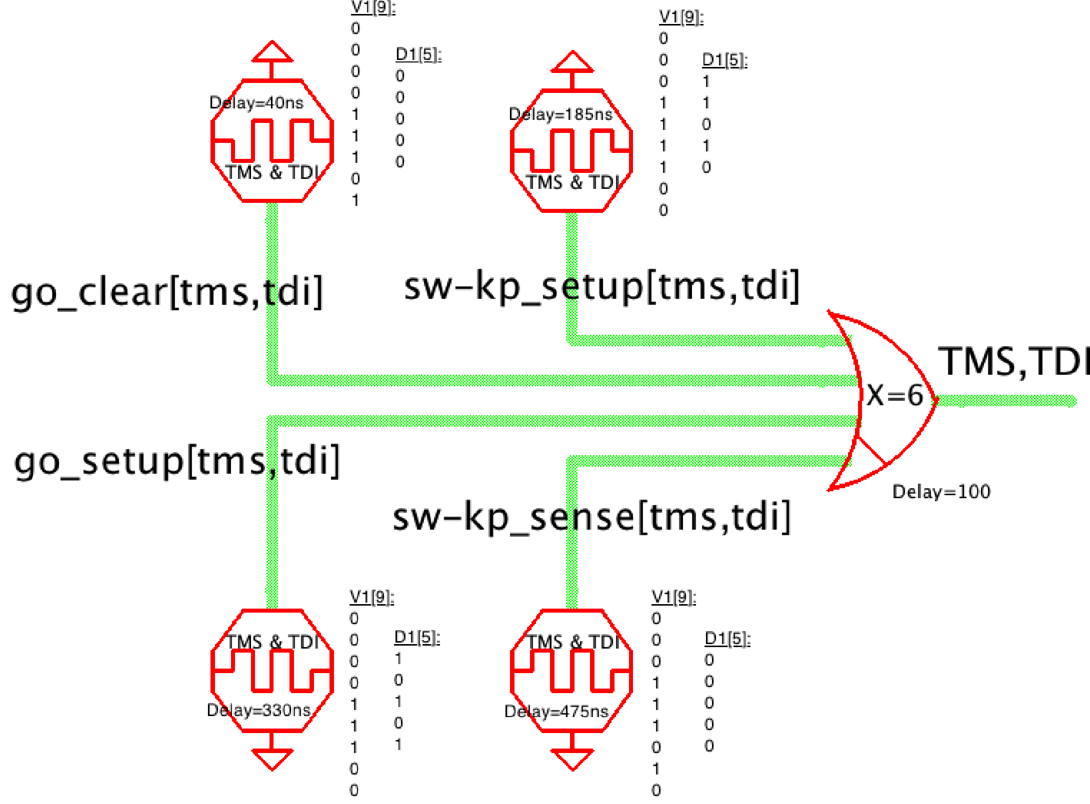


Figure 33: Details of the TMS-TDI Pulse Generators in our test program.

Pulse Name		Parameter 0: offset Delay	Delay comment	Parameter 1: V1[1:9]	Parameter 2: D1[1:5]
TCK pulse		23.5 ns	as in Figure 24	-	-
TRSTb pulse		20 ns	as in Figure 24	-	-
Scan operation	TMS-TDI Pulse				
1	go_clear [tms,tdi]	40 ns	as in **Figure 26**	000011101	00000
2	sw-kp_setup [tms,tdi]	185 ns	Delay[1] + (24 + 5) TCK cycles	000111101	11110
3	go_setup [tms,tdi]	330 ns	Delay[2] + (24 + 5) TCK cycles	000011100	10100
4	sw-kp_sense [tms,tdi]	475 ns	Delay[3] + (24 + 5) TCK cycles	000011100	10110

Table 5: Overall parameter settings of the Pulse Generators that define the test program.

6.2. Scan Operations and SPICE Simulation Results

In this Section, we explain the intent of each scan operation. To indicate the intent of the scan operations, we will describe the FIFO state changes that we expect to see as a result of the scan test operations. We will then show the 180nm SPICE simulation waveforms, and check that the reported FIFO state changes match our expectations.

We represent FIFO state as the values of its GO and statewire signals; we'll ignore the data values sent along a FULL statewire. This makes sense because this report focuses on the GasP control part of the design, and ignores the datapath.

Notation:
To describe a FIFO state, we use the notation introduced in ARC2012-is04 [4], where:

- A, B, etc **:** represents a FULL statewire, with data value A, B, etcetera
- **-** **:** represents an EMPTY statewire
- . **:** represents a GasP module with permission to act (GO=HI)
- | **:** represents a GasP module denied permission to act (GO=LO)

We additionally indicate unknown states, e.g. prior to initialization, where:

- X : represents an unknown statewire state
- * : represents a GasP module with unknown permission to act

Actions:
Each stage in the 5-stage FIFO contributes to the FIFO state changes, as follows:

- FIFO[2], FIFO[3], and FIFO[4]:
  These are instances of the GasP_plain FIFO stage design in ARC2013-smg08 [5]. Their states have three signals, a GO signal and two statewire signals, one to the previous stage and one to the next stage in the FIFO. **A module of this type changes state only when GO is HI, the previous statewire EMPTY and the next statewire FULL.** It then moves the data from the FULL to EMPTY statewire, and swaps the FULL and EMPTY indicators:
  `A.- → -.A`
- FIFO[1]:
  This is an instance of the GasP_SRC FIFO stage design in ARC2013-smg08 [5]. Its state has two signals, a GO signal and a statewire to the next FIFO stage. **The module changes state only when GO is HI and the statewire is EMPTY.** It then presents new data to the statewire, and declares the statewire FULL:
  `.- → -.A`
- FIFO[5]:
  This is an instance of the GasP_SNK FIFO stage design in ARC2013-smg08 [5]. Its state has two signals, a GO signal and a statewire to the previous FIFO stage. **The module changes state only when GO is HI and the statewire FULL.** It then removes the data from the statewire, and declares the statewire EMPTY:
  `A. → -.`

We will denote the state of the full 5-stage FIFO as the value sequence of its GO and SW signals, ordered as: GO[1] SW[1] GO[2] SW[2] GO[3] SW[3] GO[4] SW[4] GO[5].

This gives the following statement of intent for the four scan test operations:

1. **go_clear [tms,tdi]**:
   brings FIFO[1-5] from state: *X*X*X*X*
   to state: |X|X|X|X|
   This end state is confirmed by the SPICE waveforms in **Figure 34** (bottom row).

2. **sw-kp_setup [tms,tdi]**
   brings FIFO[1-5] from state: |X|X|X|X|
   to state: |-|B|-|A|
   The statewire end states are confirmed by the SPICE waveforms in **Figure 35**.

3. **go_setup [tms,tdi]**
   brings FIFO[1-5] from state: |-|B|-|A|
   to limited self-timed operation
   starting in state: .-|B.-|A.
   ending in state: .C|-.B|-.
   These state transitions are confirmed by the SPICE waveforms in Figure 36.

4. **sw-kp_sense [tms,tdi]**
   keeps FIFO[1-5] in state: .C|-.B|-.
   while serially shifting out the values of SW[4] to SW[1] at JTAG output TDO.
   State and scan values are confirmed by the SPICE waveforms in **Figure 37**.

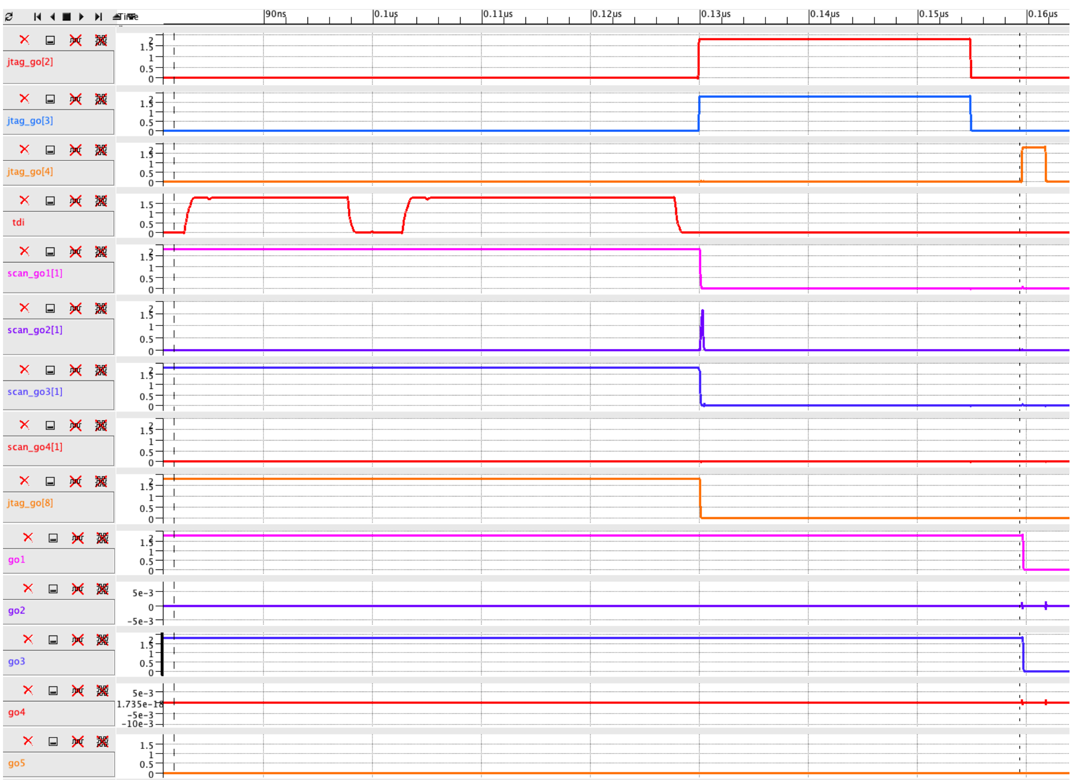


Figure 34: SPICE waveforms for scan operation go_clear [tms,tdi] with from top to bottom row (see **Figure 21**):
(1-2) the two – here overlapping – scan shift clocks, **SCCK1** and **SCCK2**
(3) GO scan chain signal **write**
(4) pulse input **TDI**
(5-9) **GO scan chain [1:5]** where all scan latches take over the LO value of TDI soon after SCCK1-SCCK2 go HI
(10-14) **GO [1:5]** copy the LO value from their scan register on the write HI pulse, thus bringing the FIFO state FIFO [1:5]
from state: *X*X*X*X*
to state : |X|X|X|X|

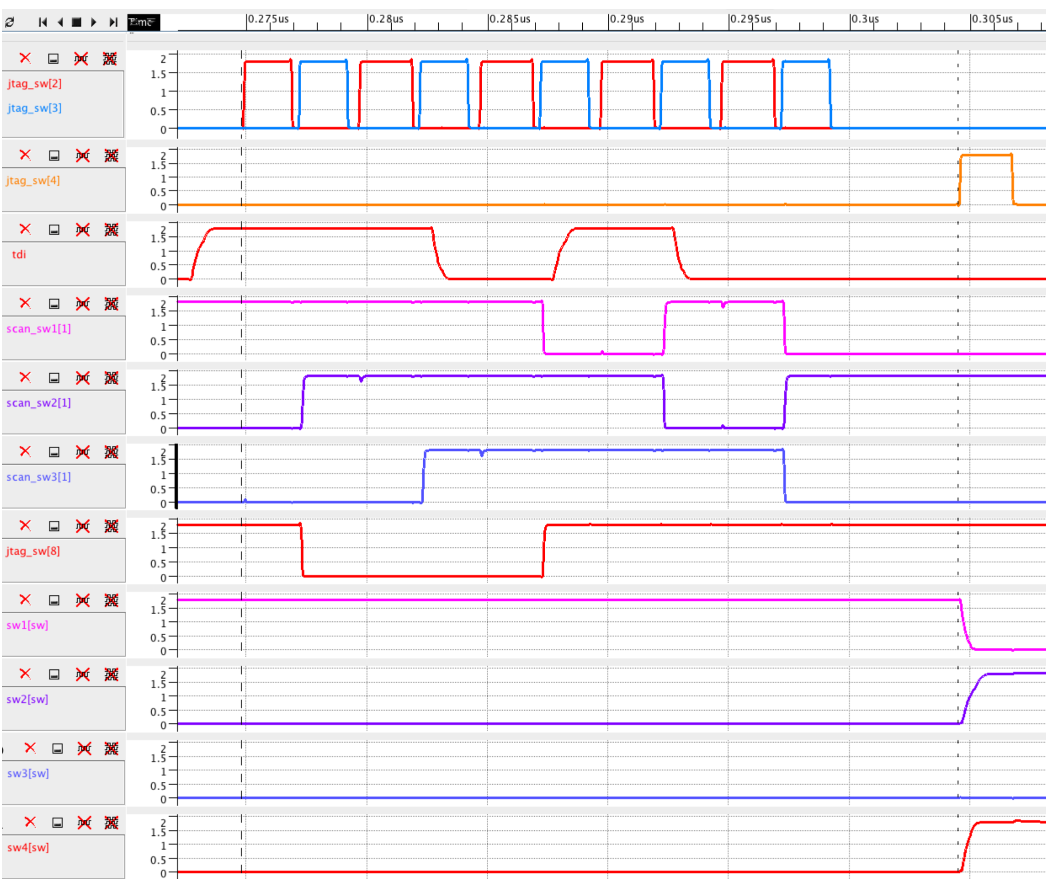


Figure 35: SPICE waveforms for scan operation sw-kp_sense [tms,tdi] with top to bottom row (see **Figure 21**):
(1) the two – here non-overlapping – scan shift clocks, **SCCK1** and **SCCK2**
(2) SW-KP scan chain signal **write**
(3) pulse input **TDI**
(4-7) **SW-KP scan chain [1:4]** is EMPTY-FULL-EMPTY-FULL at the end of the shift period, where EMPTY=LO and FULL=HI
(8-11) **SW [1:4]** copy the value from their scan register upon write HI, giving EMPTY-FULL-EMPTY-FULL, thus bringing the FIFO state FIFO [1:5]
from state: |X|X|X|X|
to state : |-|B|-|A|

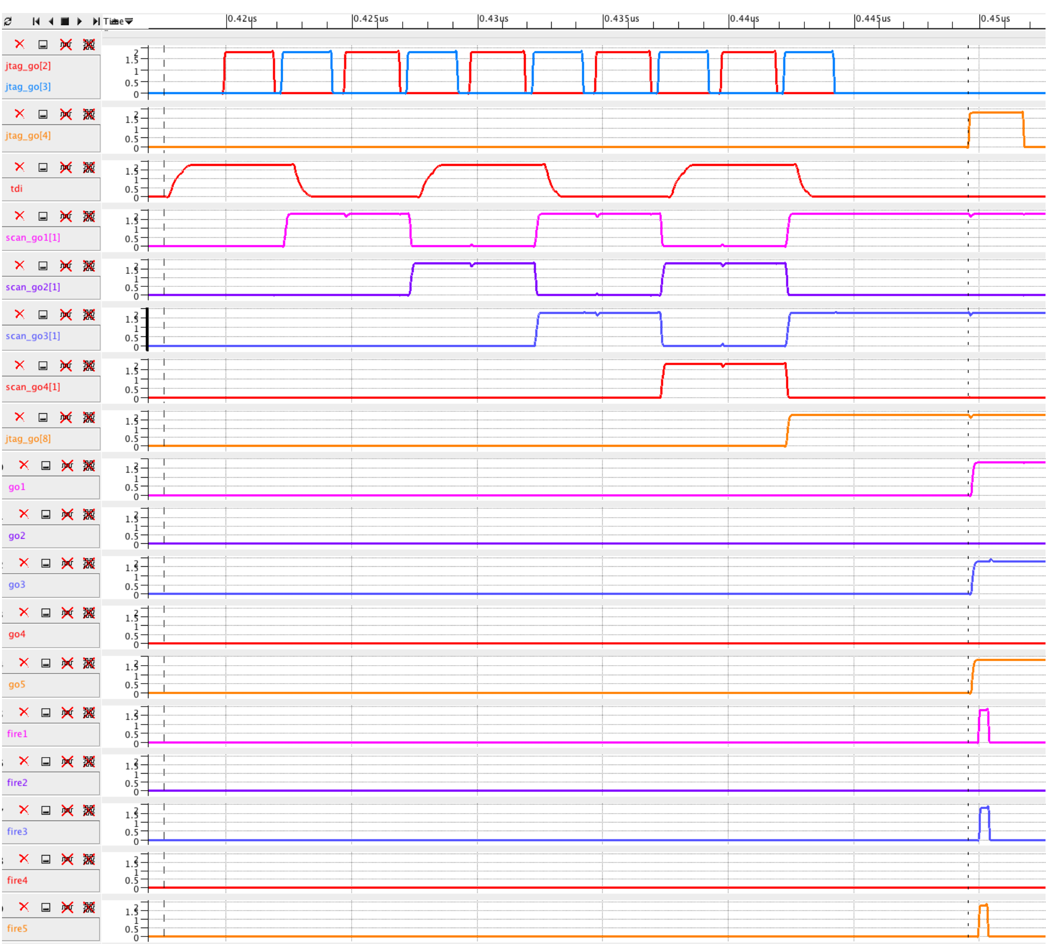


Figure 36: SPICE waveforms for scan operation go_setup [tms,tdi] with top to bottom row (see **Figure 21**):
(1) the two – here non-overlapping – scan shift clocks, **SCCK1** and **SCCK2**
(2) GO scan chain signal **write**
(3) pulse input **TDI**
(4-8) **GO scan chain [1:5]** is HI-LO-HI-LO-HI at the end of the shift period
(9-13) **GO [1:5]** copy the value from their scan register on the write HI pulse, giving HI-LO-HI-LO-HI,
thus bringing the FIFO state FIFO [1:5]
from state : |-|B|-|A|
to limited self-timed operation
starting in state : .-|B.-|A.
and ending in state : .C|-.B|-.
as will be visible in the corresponding statewire waveforms in Figure 37
(14-18) GasP generated signals **fire [1:5].**
A HI pulse on a fire[i] will flip the current state of all statewires of FIFO[i].
In this case, there's a HI pulse for fire[1], fire[3], and fire[5],
which will result in a statewire change for SW[1], SW[2] and SW[3], and SW[4], respectively.
These changes are visible in the statewire waveforms shown in Figure 37.

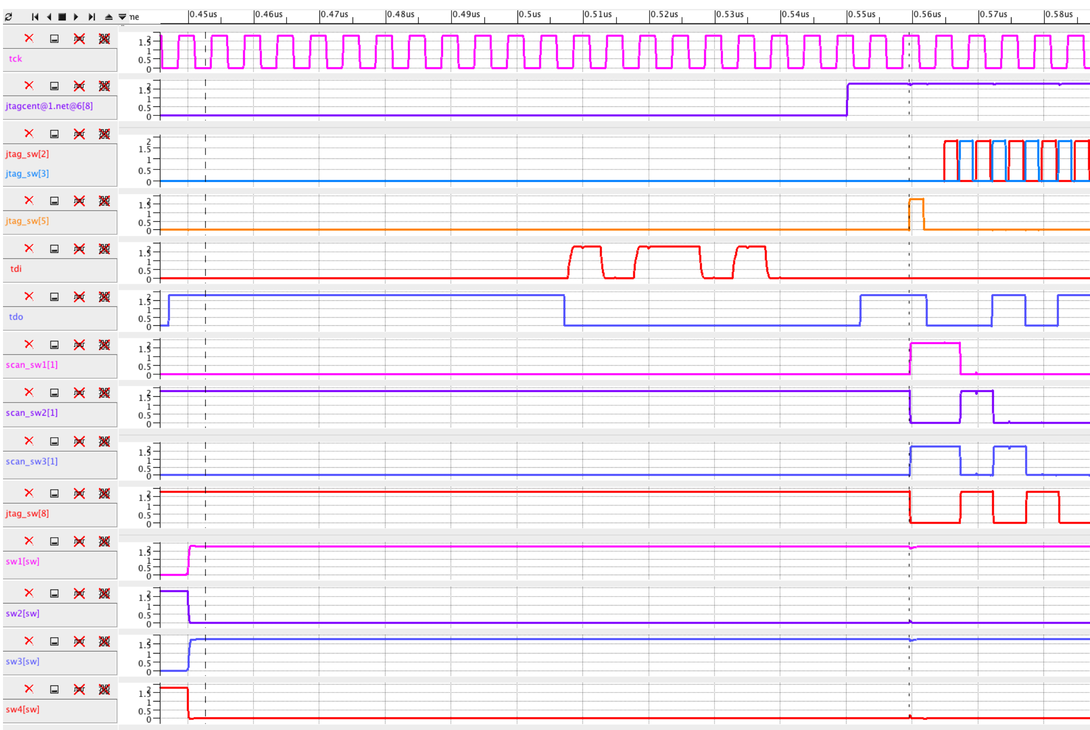


Figure 37: SPICE waveforms for scan operation sw-kp_setup [tms,tdi] with top to bottom row (see **Figure 21**):
(1) pulse input **TCK**
(2) SW-KP scan select signal **sel**
(3) the two – here non-overlapping – scan shift clocks, **SCCK1** and **SCCK2**
(4) SW-KP scan chain signal **read**
(5) pulse input **TDI**
(6) JTAG output **TDO** takes the value of SW-KP scan chain [4] at TCK LO when the SW-KP scan select signal is HI. During the five SCCK1-SCCK2 pulses, the TDO values at TCK/SCCK2 HI are: EMPTY, FULL, EMPTY, FULL.
(7-10) **SW-KP scan chain [1:4]** read the final state values of SW [1:4] during the HI read pulse, and shift these to TDO during the five successive non-overlapping SCCK1-SCCK2 HI pulses.
(11-14) **SW [1:4]** change in the self-timed operation initiated by scan operation go_setup [tms,tdi] in **Figure 36**, going from EMPTY-FULL-EMPTY-FULL to FULL-EMPTY-FULL-EMPTY, thus bringing the FIFO state FIFO [1:5]
from state : .-|B.-|A.
to state : .C|-.B|-.

7. Summary

This document follows up on ARC reports ARC2012-is04 and ARC2013-smg08 [4-5]. It is the second reports of the report pair ARC2013-smg08/smg09 with scan test solutions for Gasp circuits. Where the previous report [5] gives the details of the scan chain and the scan connections in the GasP modules, using a FIFO GasP design as reference, the current report gives the details of the JTAG controller and the programming environment that we set up to make the interactions with the JTAG controller more user-friendly. This report also shows four consecutive scan test operations that we programmed and the corresponding SPICE simulated waveforms.

Our programming environment works fine for small designs with short scan chains. For bigger designs with longer scan chains, we should really move from our, not very scalable, SPICE-level Pulse Setup to a scalable Verilog-level Pulse Setup environment. We have not yet experimented with the scan software interface developed by Oracle, but we expect that the Oracle software will come with a scalable user interface for programming scan operations.

Appendix G

Arbiter Design: Reduced Input Load

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapter 7 and can be found as reference [28] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland.
  Arbiter Design Improvements for GasP. *Technical Report, ARC2012-smg05, Asynchronous Research Center, Portland State University*, March, 2012.

Asynchronous Research Center
Portland State University

Subject: **Arbiter Design Improvements for GasP**
Date: **28 March, 2012**
From: **Swetha Mettala Gilla, Marly Roncken and Ivan Sutherland**
ARC#: **2012-smg05**

References:

[1] Jo Ebergen, Bill Coates, and Austin Lee. Long-Distance On-Chip Communication using GasP. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 119-116, 2011.
[2] Asynchronous Research Center website: http://arc.cecs.pdx.edu/.
[3] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Long-Range GasP with Charge Relaxation. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 185-195, 2010.
[4] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Telescope GasP: Overview. *ARC2012-smg01, Technical Report, Asynchronous Research Center, Portland State University,* 2012.
Note: ARC2012-smg01 has been fully integrated into Chapter 2 of Swetha's Ph.D. thesis.
[5] — Telescope GasP Storage & Broadcast: Store, Amplify, Fork and Join. *ARC2012-smg02, Ibidem.*
Note: ARC2012-smg02 is included as Appendix A in Swetha's Ph.D. thesis.
[6] — Telescope GasP Narrowcast & Arbitrated or Data Driven Control: Merge, Branch, and Distribute. *ARC2012-smg03, Ibidem.*
Note: ARC2012-smg03 is included as Appendix B in Swetha's Ph.D. thesis.
[7] — Telescope GasP: Repeat. *ARC2012-smg04, Ibidem.*
Note: ARC2012-smg04 is included as Appendix C in Swetha's Ph.D. thesis.
[8] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An Implementation Style for Data-Driven Compilation. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 3-14, 2010.
[9] Steven M. Rubin. Using the ELECTRIC™ VLSI Design System, Version 8.11. *Static Free Software and Sun Microsystems, ISBN 0-9727514-3-2,* R.L. Ranch Press, 2010.
[10] Charles Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems, Carver Mead and Lynn Conway (Eds), Addison-Wesley*, pages 218-262, 1980.
[11] Ivan Sutherland. A Mutual Exclusion Pass Gate, *ARC2010-is26, Technical Report, Asynchronous Research Center, Portland State University,* May 2010.
[12] Ivan Sutherland. Fourth Class Handout: Proper Stopper, *ARC2012-is49, Technical Report Asynchronous Research Center, Portland State University,* October 2010.
[13] Ivan Sutherland. Seventh Class Handout - Data-Controlled Branch & Demand Merge, *ARC2012-is49, Technical Report, Asynchronous Research Center, Portland State University,* November 2010.
[14] Ivan Sutherland and Scott Fairbanks. GasP: A Minimal FIFO Control. In *Proc. Int. Symposium on Asynchronous Circuits and Systems (ASYNC),* pages 46–53, 2001.

Abstract

This document analyzes our current arbiter implementation in the context of single-track asynchronous design families such as Telescope GasP [1,4,5,6,7] and traditional GasP [14,3]. During simulations of the Telescope GasP implementations for the arbitrated Merge module in [5,6,7,8], we noticed a substantial voltage drop in the arbiter selected input – see [**Figure 21**(top), 6]. We anticipated that we could diminish this voltage drop at the cost of a somewhat larger minimum latency through the arbiter. We made a new

This document contains information developed at the Asynchronous Research Center at Portland State University. Disclose this information to whomever you choose. Distribution is permitted with adequate reference to the source of the ideas and information. You may reproduce this material for any educational use. Copies of the material must contain this notice.

arbiter design with a sufficiently small voltage drop for the arbiter selected input, and used it in our second Arbitrated Merge design – see [**Figure 18**, **Figure 23** and **Figure 24**, 6]. The waveforms of the new arbiter design have relatively stable and well-defined voltage levels for the arbiter selected inputs.

In the current document, we explain the design issues of the original arbiter implementation and evaluate several possible solutions, including the new arbiter used in [**Figure 18**, **Figure 23** and **Figure 24**, 6] — which we like best. The new arbiter design can be used safely in Telescope GasP designs as well as in traditional GasP designs for 4-2 GasP [14] or 6-4 GasP [4]. It can be used safely in any other single-track asynchronous circuit family, as well as in multi-track asynchronous circuit families such as Click [8]. The design is based on the **interlock element** in [**Figure 7.25**, 10] by Charles Seitz. We discussed this design in earlier ARC reports [11,12]. The design issues addressed in the current document relate to gate and transistor topology and sizing. In addition to changing the arbiter implementation, we also evaluate solution approaches with stronger keepers and gate re-orderings.

Keywords: interlock element, arbiter, single-track handshaking, GasP, Telescope GasP.

Notation:

1. The GasP and Telescope GasP implementations that we use in this document to test the various arbiter implementations are non-inverting, i.e., their handshake channels, a.k.a. statewires, use the same encoding. Statewires say whether the data bundled with the statewire are valid. We use:
   - HI for a high voltage level, e.g. VDD, to indicate a handshake REQUEST phase with valid data.
   - LO for a low voltage level, e.g. VSS or GND, to indicate a handshake ACKNOWLEDGE phase where data are no longer needed.

   In our circuit diagrams, we use a short 45-degree line segment for VDD and a triangle for VSS. A PMOS transistor connected to VDD and an NMOS transistor connected to VSS are depicted as:

2. Our schematic designs use a so-called JBOX construct. This is a special construct in Electric [9] to connect signals with different names. The symbol for a JBOX construct is a box with the letter J in it. We use it for instance to join the **master clear (mc)** signals in the various modules. For example, **Figure 1** on page 4 uses a JBOX to connect the master clear signals **inA[mc]**, **inB[mc]**, and **out[mc]**.

Acknowledgements: We gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD thesis work.

Table of Contents

References 1
1. INTRODUCTION 3
2. Reference Arbitrated Merge and Arbiter Designs 6
3. Arbiter Implementations with Lower Input Capacitance 9
   3.1. Resizing 9
   3.2. Re-ordering the NMOS Transistor Stacks 11
   3.3. Combining Resizing with Re-Ordering the NMOS Stacks 12
4. Alternative Solution: LO Driver with Strong HI Keeper 13
5. Gate Re-ordering Solutions 17
6. Conclusion and Future Work 21

1. INTRODUCTION

Figure 1 below shows a Telescope GasP implementation for an Arbitrated Merge module. The module implementation is explained in detail in [6], and also in Section 2 of this document. The module uses an arbiter that is based on the **interlock element** [**Figure 7.25**, 10] by Charles Seitz. We discussed this type of arbiter in earlier ARC reports [11,12]. All our arbiter designs are based on it.

We simulated the Arbitrated Merge module with two different arbiter implementations. The arbiter implementations *are the same* in the sense that both:

1. arbitrate,
2. hide metastable voltage levels until these are resolved, and
3. alternately select **inA[sw]** followed by **inB[sw]**, or vice versa, whenever these two inputs overlap.

The two arbiter implementations *differ* in topology and transistor sizes, and as result, they have different latency and input capacitance.

These differences in latency and input capacitance are reflected in the waveforms for statewires **inA[sw]**, **inB[sw]**, and **out[sw]** in **Figure 2**(top) respectively **Figure 2**(bottom). Note that:

1. The cycle period for **out[sw]** in the top simulation window of **Figure 2** is about 0.75 nanoseconds or 750 picoseconds. The cycle period for **out[sw]** in the bottom window is about 825 picoseconds, i.e., 10% higher than the cycle time recorded in the top simulation window. This is because the latency through the arbiter implementation used for the bottom simulation window is larger than the latency through the arbiter implementation used in top simulation window.
2. The HI voltage for the two statewires of the incoming channels **inA[sw]** and **inB[sw]** drops to about 1.4 Volt, 75% of VDD, in the top window and to somewhere between 1.7 and 1.75 Volt, 95% to 97% of VDD, in the bottom window. In each case, the voltage drop occurs when the arbiter grants the incoming channel the right to proceed. In each case, the voltage bounces back slowly to HI due to the keeper operation on the statewire.

We are not happy with the voltage drop in the top window: it makes the statewires more susceptible to noise and signal interference than we're comfortable with. This is an issue especially in situations where the arbiter is kept and not released for a long time, as can easily happen in Telescope GasP designs.

The two simulations in the top and bottom windows of **Figure 2** reflect what we did to tackle the Voltage drop issue: we diminished the HI voltage drop by replacing the original arbiter with a new implementation, at the cost of a somewhat larger minimum latency in the new arbiter.

In the rest of this document, we will explain in detail the differences between the two arbiter implementations. We will explore intermediary arbiter solutions as well as other solutions such as statewire drivers with strong HI keepers to diminish the HI voltage drop on the wire. We use the Arbitrated Merge of **Figure 1** to simulate the effects in latency and HI voltage drop for these various solutions.

The rest of the document is organized as follows. Section 2 discusses the key behaviors and design features of our reference Arbitrated Merge and arbiter designs. Section 3 evaluates the latency and input capacitance of various arbiter implementations. For each arbiter implementation, we evaluate the cycle time as seen by the Arbitrated Merge module of **Figure 1** with the arbiter, and we evaluate the HI voltage drop on the handshake channels coming into the Arbitrated Merge. In Section 4, we look at alternative design solutions. Rather than changing the arbiter implementation to lower the input capacitance on the statewires, instead we change the implementation of the statewire drivers by strengthening the HI keeper in the driver so it can cope with a high capacitance arbiter. In Section 5, we look for solutions to shield a statewire from a high capacitance arbiter by re-ordering the gates in the module so that a low capacitance gate is placed between the statewire and the arbiter. Section 6 summarizes the results and concludes this document.

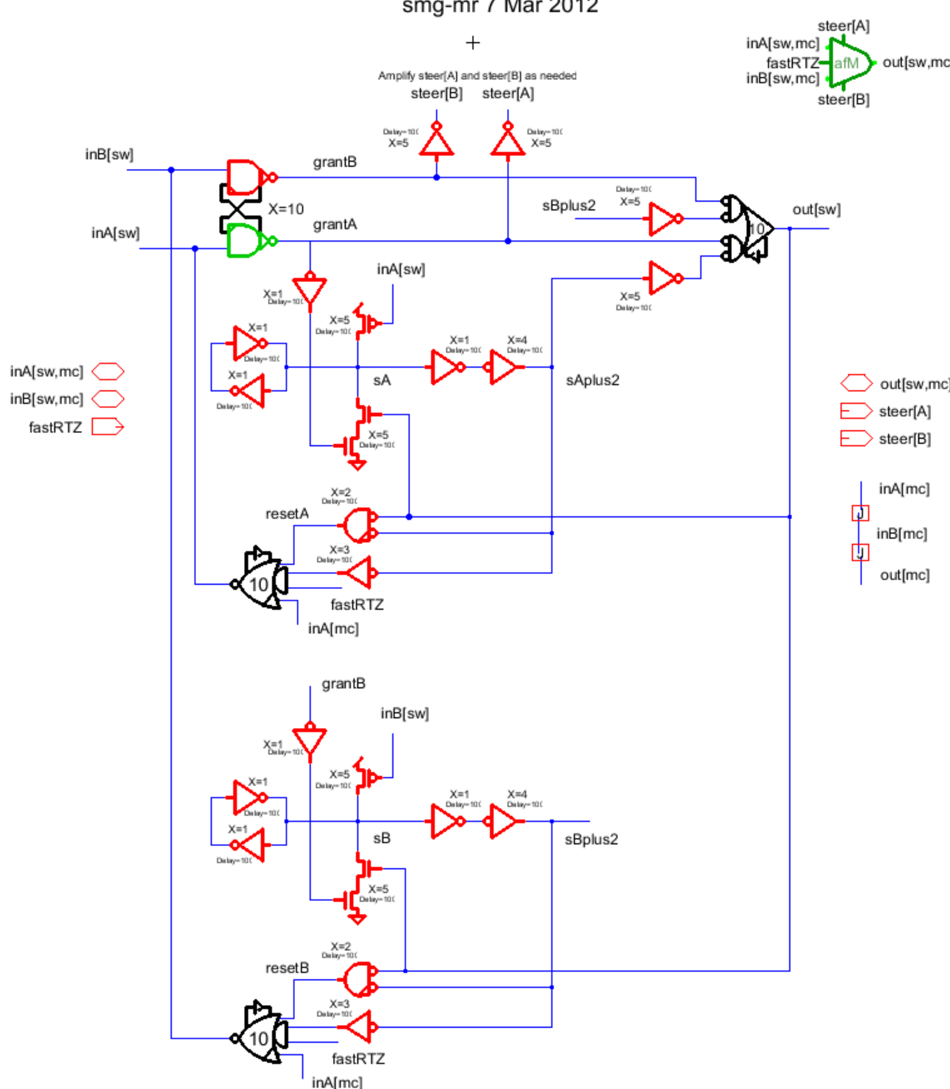
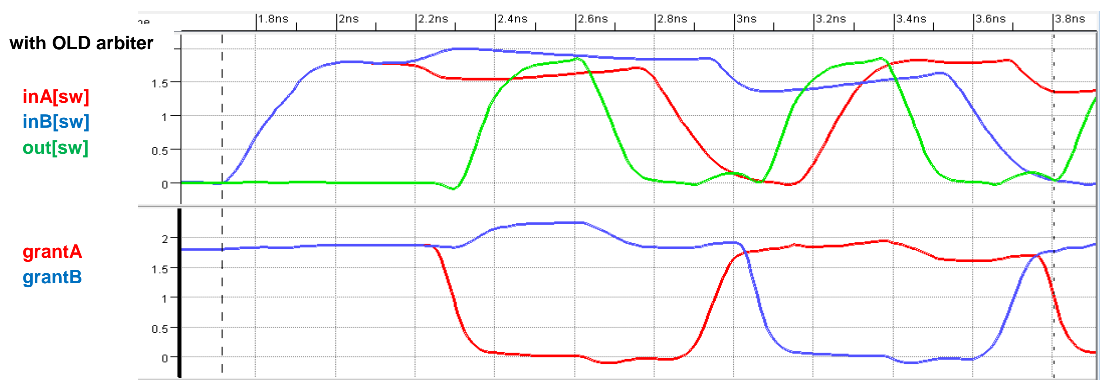


Figure 1: Telescope GasP implementation for an Arbitrated Merge module. The arbiter is represented by the pair of cross-coupled NAND gates. The Arbitrated Merge module has two input statewires with a master clear signal, **inA[sw,mc]** and **inB[sw,mc]**, two corresponding data steering signals **steer[A]** and **steer[B]**, and one output statewire with a forwarded master clear, **out[sw,mc]**. We will use the icon in the top-right corner of this picture, as a gate-level representation of an Arbitrated Merge module. This particular module shares the self-resetting loops between incoming and outgoing statewires. It is saved in Electric's ARC library TelescopeGasP_wStateSharing under module name Merge2TfastRTZwArbiter. `TfastRTZ' in the module name indicates that this is a Telescope GasP implementation with fast reset capability. To ensure that we reset only the incoming handshake that caused **out[sw]** to go HI, we have AND-ed the fast reset signal at the LO driver with an internal guard signal. The guard for **inA[sw]** is tapped from internal state signal **sA** via three inversions, The guard for **inB[sw]** is tapped from internal state signal **sB** via three inversions. Fast resetting reduces the cycle time and backward latency of Telescope GasP designs. For more details, see [6].



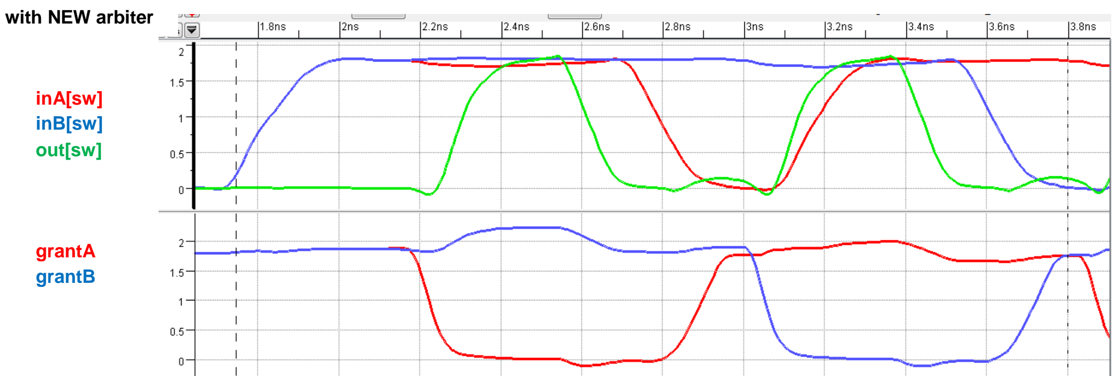


Figure 2: SPICE simulations of the Arbitrated Merge in **Figure 1** with original arbiter implementation (top) and new arbiter implementation (bottom). Red and blue signals inA[sw] and inB[sw] have a telescopic handshake relationship to green signal out[sw]: inA[sw] or inB[sw] HI causes out[sw] to go HI, and when out[sw] has gone LO, only then will the corresponding input handshake signal go LO. The simulation clearly shows that each HI out[sw] pulse is contained within a HI pulse on inA[sw] or inB[sw]. Both windows show the fairness property of the arbiter: when red signal inA[sw] and blue signal inB[sw] arrive at about the same time and the arbiter selects the red inA[sw], then it will select the blue inB[sw] next.

- **(top)** We are not happy with the voltage drop in the arbiter selected input, visible in the top window. The HI incoming handshake communication on inA[sw] respectively inB[sw] drops from 1.8 to 1.4 Volt whenever the handshake is granted, i.e., whenever grantA respectively grantB goes LO. We can diminish this voltage drop at the cost of a somewhat larger minimum latency through the arbiter, as shown in the bottom simulation window. The current cycle time for out[sw], as measured for instance from the time out[sw] is at 1 Volt to the next time that out[sw] is at 1 Volt, is 750 picoseconds.
- **(bottom)** With the new arbiter, the HI voltage for inA[sw] respectively inB[sw] in the bottom window drops from 1.8 to about 1.7 Volt, whenever grantA respectively grantB goes LO. The cost for improving the voltage levels for inA[sw] and inB[sw] is a reduction in cycle time. The new cycle time for out[sw] in the bottom window is 825 picoseconds. This is 10% slower than the cycle time for out[sw] in the top window.

2. Reference Arbitrated Merge and Arbiter Designs

Figure 1 shows our reference Telescope GasP design for the Arbitrated Merge module in the context of this document. We currently use two implementation styles for Telescope GasP modules, as explained in [5,6,7,8]. The two styles behave the same regarding arbitration functionality and timing. We can therefore select arbitrarily one of the two Arbitrated Merge designs from [6] as reference design for evaluating different arbiter implementations. The Arbitrated Merge in **Figure 1** has shared self-resetting loops between the statewires of incoming and outgoing handshake channels. It has two self-resetting loops, with separate taps to start and stop the HI drive for **out[sw]** and the LO drives for **inA[sw]** and **inB[sw]**.

A SPICE-level simulation of the handshake behavior of the Arbitrated Merge implementation was given earlier in **Figure 2**, for two different implementations of the arbiter. Though timing and voltage levels differ, the handshake behaviors are essentially the same for both arbiter implementations and can be described as follows. Initially, **inA[sw]**, **inB[sw]**, **out[sw]**, and **mc** are LO. When either or both of **inA[sw]** and **inB[sw]** go HI, indicating the presence of data on the incoming channels, and when **out[sw]** is LO, indicating the availability of space, the following four sets of actions are started **in sequence**, as follows:

1. Indicate the presence of new data to the successor module by driving **out[sw]** HI. This takes 2 gate delays. Meanwhile, drive the steering signal for the arbiter-selected incoming channel HI.
2. Cut off the HI forward drive to a 5 gate delay pulse signal.
3. Wait until **out[sw]** is LO, and then report new availability of space to the predecessor module by driving the arbiter-selected incoming channel LO, 2 gate delays after observing the LO **out[sw]**. Meanwhile, also reset the forwarded data steering signals to LO.
4. Cut off the LO backward drive to a 5 gate delay pulse.

After action 4, the Arbitrated Merge waits until either **inA[sw]** or **inB[sw]** is HI, so it can start its next cycle.

The minimum forward latency in the Arbitrated Merge is 2 gate delays: 1 through the arbiter and 1 through the successor driver. This minimum forward latency holds if the arbiter is uncontested, for instance when there is only one input request. The arbiter is a metastable device and so, in theory, when both input requests arrive at about the same time, the delay through the arbiter can be arbitrarily long. In practice, however, arbitrations are rarely contested, and their resolution time is generally around 1 gate delay.

Without a fast reset signal, the minimum cycle time for the Arbitrated Merge module is 2+5+2+5=14 gate delays, as set by the 2 gate delays for forward and backward transfers and the 5 gate delay self-resetting loops in the circuit diagram of **Figure 1**. The cycle time increases in steps of 4 gate delays for each successor module between the Arbitrated Merge module and the following Store module. Using a fast reset signal, we can reduce the backward latency and lower the cycle time to 12 gate delays per Arbitrated Merge module, with an increase in steps of 2 gate delays for each successor module between the Arbitrated Merge module and the following Store module. For details, see [6].
.
Figure 3 and **Figure 4** show a hierarchical schematics of our original arbiter design - see also [6]. The design is based on the **interlock element** in [**Figure 7.25**, 10] by Charles Seitz – see also [11,12]. The organization in **Figure 3** has been proven topologically equivalent to the design by Seitz, using Electric [9]. In our arbiter evaluation, we will work with the flattened version, shown in **Figure 5**. Each of our arbiter implementations is fair in that it will alternately select **inA[sw]** followed by **inB[sw]**, or vice versa, whenever these two inputs overlap. This is because it will take at least 5 gate delays for the selected input statewire to bounce back from LO to HI, while the unselected input statewire is already HI.

Figure 6(top) repeats the external handshake behavior of the arbitrated Merge module shown earlier in **Figure 2**(top). **Figure 6**(bottom) shows a SPICE simulated arbitrated resolution of two competing arbiter inputs and the corresponding changes in voltage levels for intermediate and output signals of the arbiter. We are not happy with the 1.8 Volt to 1.4 Volt voltage drop in the arbiter selected inputs in **Figure 6** (top). We can diminish this voltage drop at the cost of a somewhat larger minimum latency through the arbiter. Solutions for diminishing the input voltage drop by changing the arbiter implementation follow in Section 3.

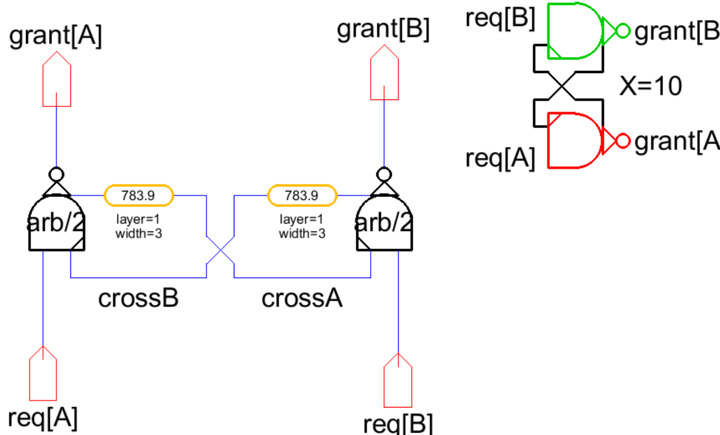


Figure 3: Hierarchical schematics for the 2-way arbiter, called Arbitrated Latch. The Arbitrated Latch is implemented as a cross-coupled pair of so-called Arbitrated Latch Nand gates, whose design follows below in **Figure 4**. When one or both of the requesting inputs **req[A]** and **req[B]** are HI, the arbiter grants exactly one of the HI requests by lowering the corresponding grant output. It lowers **grant[A]** if it grants a HI **req[A]** request, and it lowers **grant[B]** if it grants a HI **req[B]** request. It never lowers both **grant[A]** and **grant[B].** We use the top-right icon as a gate-level representation for this 2-way arbiter design.

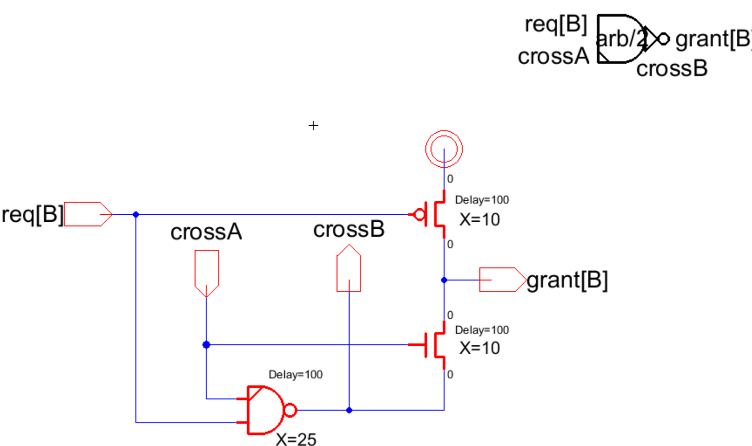


Figure 4: Implementation and icon for the Arbitrated Latch Nand circuit used in the arbiter design of **Figure 3**. Input **req[B]** is one of the input requests to the arbiter. Input **crossA** is cross-coupled to the intermediary output signal of the complementary Arbitrated Latch Nand gate in the arbiter. Likewise, output **crossB** is cross-coupled to the intermediary input signal of the complementary Arbitrated Latch Nand gate in the arbiter. Output signal **grant[B]** is driven HI whenever **req[B]** is LO. If **req[B]** is HI the cross-coupling works like an arm-wrestling match between the requests. Due to the extra voltage build-up across the NMOS pass transistor between **crossB** and **grant[B]**, even a mid-way voltage tie between **crossA** and **crossB** leaves both grant signals at a sufficiently high voltage level. The combination of cross-coupling and NMOS transistor ensures that the outgoing grant signal, here **grant[B]**, retains a high voltage level until the arbitration process resolves in favor of **grant[B]**. We believe that such a high voltage level can be retained for at least 50 gate delays. To ensure that very long arbitration periods are supported, one might consider adding a keeper on both grant[A] and grant[B]. Our current arbiter implementations do not include such keepers on the grant signals.

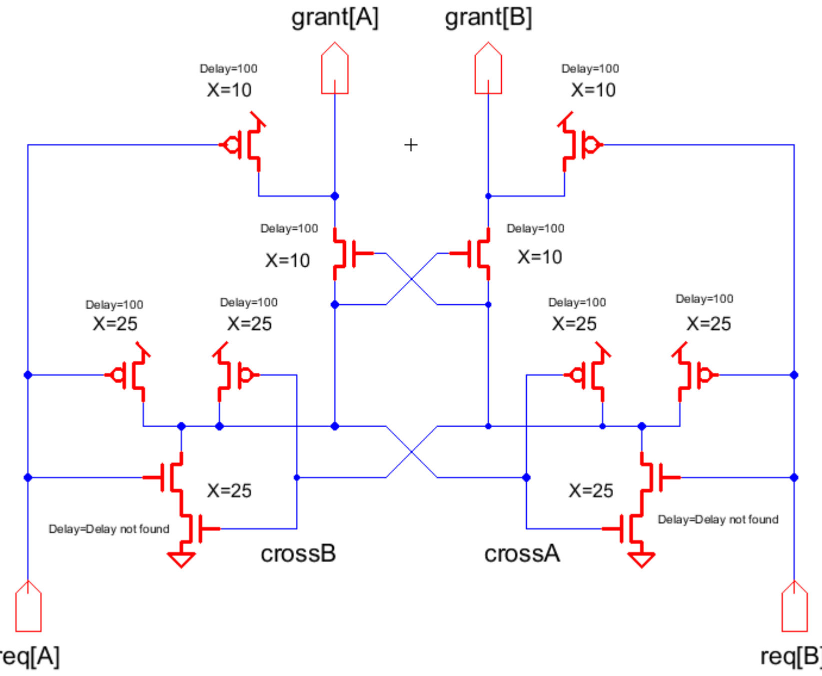


Figure 5: Flattened version of the hierarchical arbiter implementation presented in **Figure 3** and **Figure 4**.

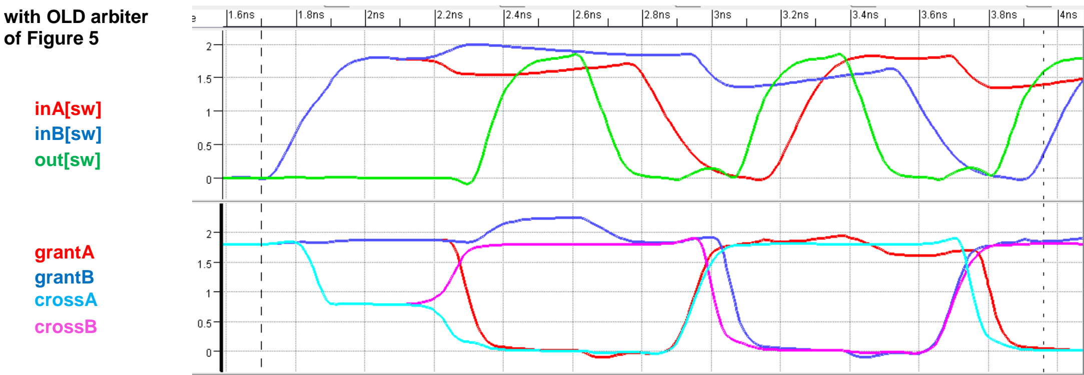


Figure 6: SPICE simulations of (top) the Telescope GasP implementation of the Arbitrated Merge in **Figure 1,** and (bottom) the resolution of two competing arbiter inputs and corresponding changes in voltage levels for intermediary signals and output signals of the arbiter design in **Figure 5**, or equivalently **Figure 3**-**Figure 4**.

- In the top window, one can see the fairness property of the arbiter: when statewires inA[sw] and inB[sw] arrive at about the same time and the arbiter selects inA[sw], then it will select inB[sw] next.
- The minimum cycle time for out[sw] as measured for instance from the time out[sw] is at 1 Volt to the next time that out[sw] is at 1 Volt, is 750 picoseconds.
- The bottom window gives a more detailed view of such a scenario, including the initial metastable behavior where the arbiter is still deciding whether to grant inA[sw] or inB[sw].
- We are not happy with the voltage drop in the arbiter selected input, visible in the top window. The voltage of inA[sw] and inB[sw] drops to about 1.4 Volt. We can diminish this voltage drop at the cost of a somewhat larger minimum latency through the arbiter, which we do in the next Section of this document.

3. Arbiter Implementations with Lower Input Capacitance

The simulations in **Figure 2** and **Figure 6** indicate that the high input capacitance of the original arbiter design in **Figure 5** causes a significant voltage drop on its inputs. For the Arbitrated Merge module in **Figure 1**, these inputs are the bidirectional statewires of handshake channels coming into the module. As a result, the HI 1.8 Volt voltage on a statewire drops by 25% to 1.4 Volt whenever the incoming channel is granted the right to proceed.

A 25% voltage drop makes the statewires more susceptible to noise and signal interference than we're comfortable with. After all: the statewires are the longest control wires in the design, because they form the handshake communication interface between the modules. This is an issue especially in situations where the arbiter is kept and not released for a long time. Telescope GasP modules have cumulative cycle times that grow linearly with the depth of the data path, as explained in [4,5,6,7]. As a result, the arbiter in the Arbitrated Merge module may be kept and not released for a long time. We must find a way to reduce the 25% voltage drop on the statewires to within say 90-100%.

Our current Telescope GasP modules are designed for small forward and backward latencies of 2 gate delays each. This low-latency design criterion leaves no room for re-ordering the gates in the forward path. The path necessarily goes through the arbiter first and through the successor driver next. As a result, the statewires go straight into the arbiter, and experience its full input capacitance.

For designs like our current Telescope GasP modules, where statewires have no choice but to go straight into the arbiter, we will discuss two solution approaches to help the statewires cope better with the input capacitance of the arbiter. The first solution approach is to re-implement the arbiter so it has lower input capacitance. That's what we do in Sections 3.1 to 3.3 below. The second solution approach is to increase the drive strength of the keepers that maintain the HI voltage level on the statewires when the predecessor driver has ceased its HI drive. Section 4 discusses the second solution approach.

Solution approaches for designs where gate re-ordering is an option are discussed in Section 5.

3.1. Resizing

The new arbiter solution that we simulated in **Figure 2**(bottom) combines two optimization steps that lower the input capacitance. The first optimization step is to reduce the size of transistors that are driven directly by inputs and that do not affect the drive strength of the arbiter. This concerns the two transistors marked **PMOS-A2** and **PMOS-B2** in **Figure 7**. By reducing the size of these two transistors, the input capacitance on the statewires of the incoming request channels is reduced to the extent that each statewire now requires a step-up-of-3 driving gate of size ((50 * (1 / 3)) + (10 * (2 / 3)) + (10 * (2 / 3))) / 3 = 10 instead of ((50 * (1 / 3)) + (25 * (2 / 3)) + (10 * (2 / 3))) / 3 = 13.3 to guarantee good slopes on **req[A]** and **req[B]**.

We re-simulated the Arbitrated Merge module in **Figure 1**, using this resized arbiter, to quantify to what extent this first optimization helps reduce the voltage drop on the statewires of the incoming handshake channels, and what it costs in cycle time. The SPICE-level waveforms are presented in **Figure 8**. The top window in **Figure 8** shows that the new cycle time for **out[sw]** is a bit larger: around 775 picoseconds. It also shows that the statewires still experience a voltage drop from 1.8 Volt to 1.4 Volt.

We conclude that this first optimization is not good enough, because the statewires still experience a voltage drop to about 75% of VDD for HI voltage levels.

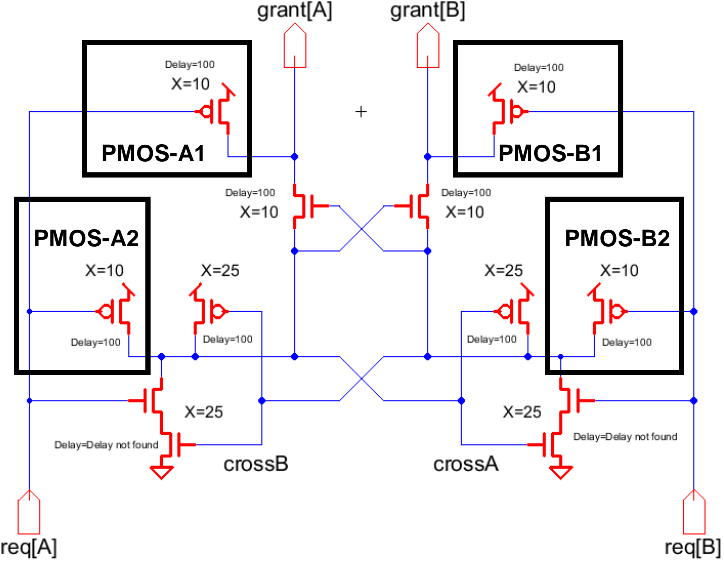


Figure 7: Resized implementation of the flattened arbiter in **Figure 5**. We kept the sizes for the two PMOS transistors at the top to maintain the HI grant drives and for the NMOS transistors at the bottom to maintain the LO grant drives. We reduced the sizes of the two supporting PMOS transistors in the middle that are connected to an incoming request signals: **PMOS-A2** and **PMOS-B2**. These two PMOS transistors help with returning a grant signal to HI at the end of the granted request phase. For instance, the left-hand **PMOS-A2** plays the supporting role of raising the voltage level on internal signal **crossA** to (1) reduce the voltage conflict over the left-hand pass-transistor connection to **grant[A]** and the on-turning **PMOS-A1** connection to VDD, and (2) enable the right-hand pass-transistor for grant[B]. We anticipate that this size reduction from 25 to 10 for **PMOS-A2** and **PMOS-B2** will have the following pros and cons:

- **Pro:** This lowers the input capacitance and thus reduces the voltage drop on the inputs - somewhat.
- **Con:** This slows down the release of the arbiter, and thus increases the cycle time - somewhat.

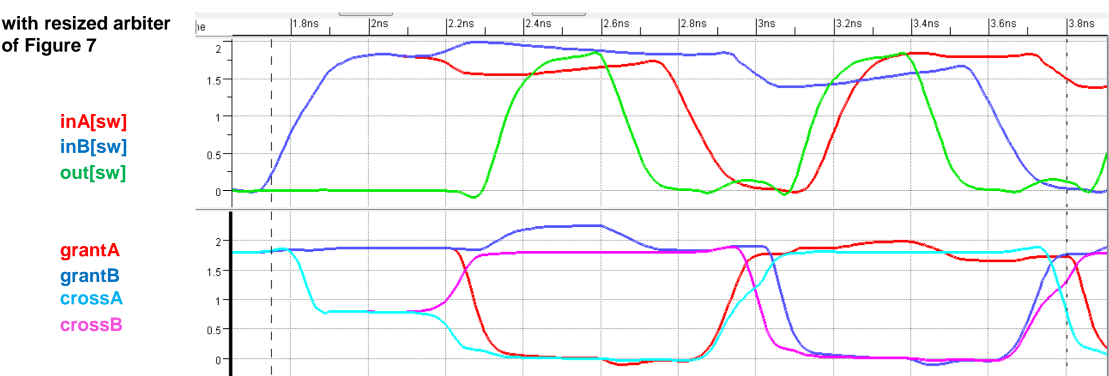


Figure 8: SPICE simulations of (top) the Telescope GasP implementation of the Arbitrated Merge in **Figure 1** with the resized arbiter of **Figure 7** and (bottom) the resolution of two competing arbiter inputs and corresponding changes in voltage levels for intermediary signals and output signals of the resized arbiter. The simulation results are similar to the results in **Figure 6** for the Arbitrated Merge with the old arbiter:

- The minimum cycle time for out[sw] as measured from the time out[sw] is at 1 Volt to the next time that out[sw] is at 1 Volt, is 775 picoseconds - 3% slower than the cycle time with the old arbiter in **Figure 5**.
- We are still not happy with the voltage drop in the arbiter selected input, visible in the top window. The voltage for inA[sw] and inB[sw] drops to about 1.4 Volt, just like before with the old arbiter.

3.2. Re-ordering the NMOS Transistor Stacks

The new arbiter solution that we simulated in **Figure 2**(bottom) combines two optimization steps that lower the input capacitance. We tried the first optimization step in Section 3.1 above, and found it insufficient. Here, we'll try just the second optimization step. The second optimization step re-orders the NMOS transistor stack, so that the request signal now drives the NMOS transistor that's closest to the VSS rail, as indicated in **Figure 9**. This will increase the latency of uncontested requests, but shield the incoming request signal better from internal voltage changes on the cross-coupled signal wires inside the arbiter. We re-simulated the Arbitrated Merge module in **Figure 1** using this new arbiter implementation, to quantify how much this second optimization step helps reduce the voltage drop on the incoming channels, and what it costs in cycle time. The SPICE waveforms follow in **Figure 10** and show a cycle time for **out[sw]** of about 775 picoseconds, and a voltage drop on the statewires down to between 1.65-1.70 Volt, 92%-95% of VDD, for HI voltage levels. This may be a workable solution. But it still requires a step-up-of-3 driving gate of size ((50 * (1 / 3)) + (25 * (2 / 3)) + (10 * (2 / 3))) / 3 = 13.3 to guarantee good slopes on req[A] and req[B].

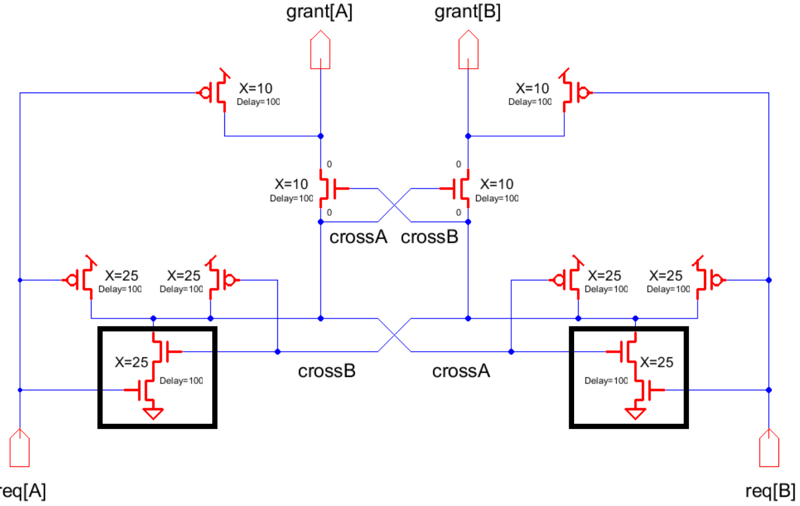


Figure 9: New arbiter implementation adapted from the flattened arbiter in **Figure 5** by swapping the two transistors in each NMOS stack. We kept the transistor sizes the same as in the old arbiter in **Figure 5**.

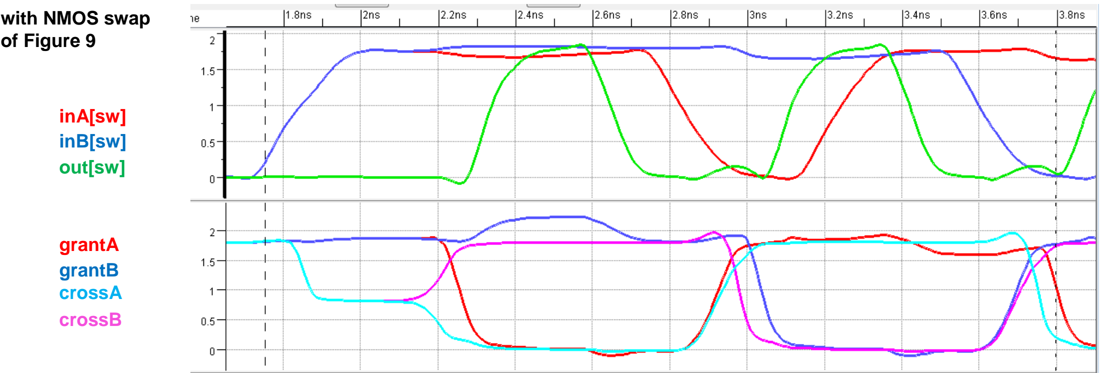


Figure 10: SPICE simulations of (top) Telescope GasP implementation of the Arbitrated Merge in **Figure 1** with the arbiter of **Figure 9 ,** and (bottom) resolution of competing arbiter inputs and corresponding changes in voltage levels for intermediary and output signals of the arbiter. The waveforms indicate (1) a cycle time for out[sw] of 775 picoseconds and (2) a voltage drop for inA[sw] and inB[sw] from 1.8 to 1.7 Volt - which is acceptable.

3.3. Combining Resizing with Re-Ordering the NMOS Stacks

The new arbiter solution that we simulated in **Figure 2**(bottom) combines two optimization steps that lower the input capacitance. We tried the first optimization step in Section 3.1 above, and found that it was insufficient. We tried the second optimization step by itself in Section 3.2 above, and found it workable. Here, we combine both optimization steps, to arrive at the solution that we like best. The new arbiter implementation and the new simulation results follow in **Figure 11** and **Figure 12**. The reduced input capacitance on req[A] and req[B] reduces the step-up-of-3 driving gate to a size of 10 instead of 13.3 to guarantee good slopes on the incoming request signals. The HI voltage drop on the statewires is down to somewhere between 1.70 and 1.75 Volt or 95%-97% of VDD - good enough to withstand external noise. Note that the voltage bounces back completely before the statewire is reset to LO by the predecessor drivers in the module; this is due to the statewire keepers. The new cycle time for **out[sw]** is approximately 825 picoseconds — 10% slower than the original cycle time of 750 picoseconds in **Figure 2**(top), which is a reasonable price to pay for the reduced input capacitance and an input voltage level between 95%-100%.

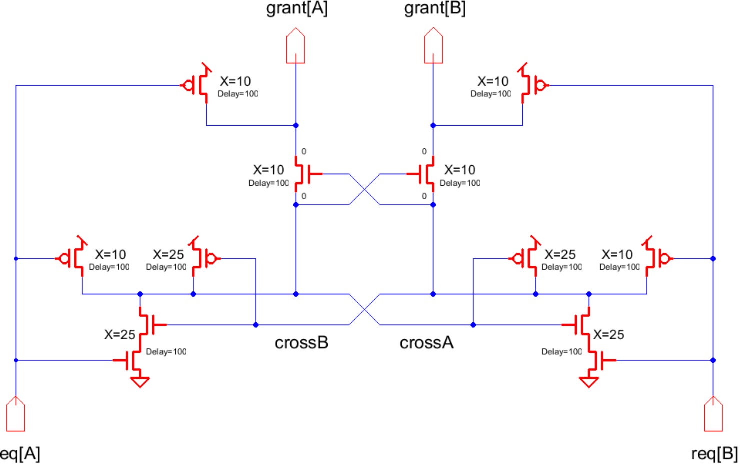


Figure 11: This is the new arbiter re-implementation of the flattened arbiter in **Figure 5** that we like best. It combines the two arbiter optimization approaches from **Figure 7** and **Figure 9**.

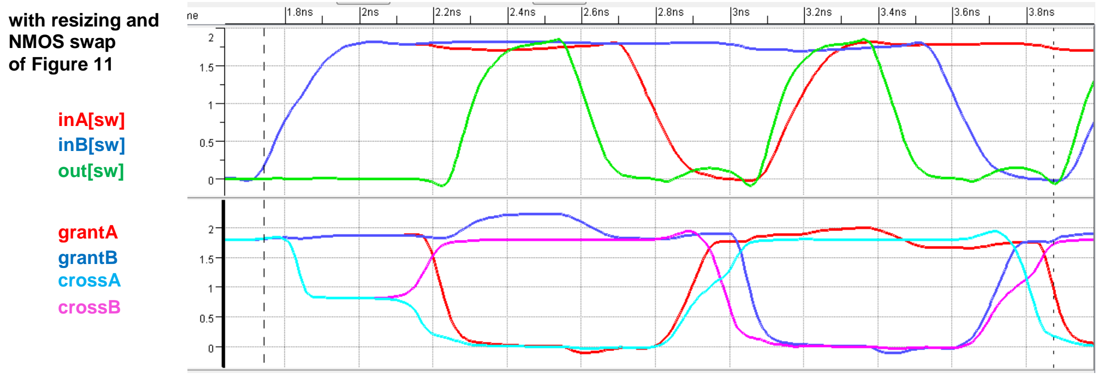


Figure 12: Same SPICE-level simulation waveforms as in **Figure 2**(bottom), but with more waveform details for the internal arbiter signals. With this new arbiter, the HI voltage for inA[sw] respectively inB[sw] drops from 1.8 to somewhere between 1.70 and 1.75 Volt, whenever grantA respectively grantB goes LO. This drop is small enough to not cause issues with noise, and also small enough for the keepers to restore the voltage to 1.8 Volt before the end of the handshake cycle. The cost for improving the voltage levels for inA[sw] and inB[sw] is a reduction in cycle time. The new cycle time for out[sw] is 825 picoseconds - 10% larger than in **Figure 2**(top) with the original arbiter implementation of **Figure 5**. We are satisfied with these final gain-loss percentages.

4. Alternative Solution: LO Driver with Strong HI Keeper

In Section 3, we looked at three solutions to reduce the voltage drop for the statewire inputs to the arbiter. These solutions involved changing the size and ordering of transistors inside the arbiter implementation. In this Section, we will look at solutions outside the arbiter, and reduce the voltage drop for the statewire inputs to the arbiter by changing the strength of the statewire drivers and keepers.

To drive the original, unchanged arbiter implementation in **Figure 5**, each input requires a step-up-of-3 driving gate of size ((50 * (1 / 3)) + (25 * (2 / 3)) + (10 * (2 / 3))) / 3 = 13.3, say 13. This drive size is needed to guarantee decent input slopes. The statewires feeding the arbiter in the Arbitrated Merge of **Figure 1** are driven HI respectively LO for about 5 gate delays by strong drivers. When these drivers are off, HI respectively LO keepers maintain the voltage level on the statewires. During arbitration, either the HI driver for the statewire is on or the HI keeper is on, or both. To satisfy the strong drive needs during arbitration, it would help if both the HI driver and the HI keeper have a drive size of 13.

The HI driver resides in the successor driver of the preceding module, and has a drive size of 10, which is sufficiently close to 13. The HI keeper resides in the predecessor driver of the Arbitrated Merge module itself. **Figure 13** gives the schematics of the predecessor driver with its LO driver and HI keeper. The HI keeper is organized as a 4-deep stack of PMOS transistors of size 4 each. Four PMOS transistors of size 4 in series give a combined size of 1. This is about an order of magnitude too small to compensate for strong drive needs during arbitration. For four PMOS transistors in series to offer a combined drive size of about 13.3, each transistor must have a size of about (13)*4=52. The new predecessor design with upsized keeper follows in **Figure 14**.

We re-simulated the Arbitrated Merge module in **Figure 1**, using this new predecessor LO driver with strong HI keeper implementation from **Figure 14**, to quantify to what extent this alternative solution helps reduce the voltage drop on the statewires of the incoming handshake channels, and what it costs.

The SPICE-level waveforms follow in **Figure 15**. They show a cycle time for **out[sw]** of approximately 900 picoseconds, 17% slower than in **Figure 2**(top) with the old arbiter and the original driver of **Figure 13**, and slower than our best arbiter result in **Figure 2**(bottom).

The voltage levels are excellent for competition-free arbitrations: the voltage for the granted incoming statewire drops to between 1.70 Volt and 1.75 Volt, and bounces back almost immediately. This exceeds the excellent results in **Figure 2**(bottom) and **Figure 12** that we obtained with our best arbiter implementation in **Figure 11**. However, for competing arbitration requests, when the arbiter needs time to make a decision, the voltage levels of both statewires rise slowly after reaching 1 Volt, and they stay around 1.4 Volt for some time, as we can see happening during the first arbitration cycle in **Figure 15**.

Also, laying out four large PMOS transistors of size 52 in series is not a solution we'd go for easily, because it makes the cell layout awkward and big, and significantly increases the logical effort of the predecessor driver. The increase in logical effort imposes stronger drive requirements for the upstream gates in the Arbitrated Merge, as we can see in **Figure 16**. Even if we would reduce the keeper size, the voltage issues during a competing arbitration request will remain.

In summary:
We prefer the new arbiter solution of **Figure 11** in Section 3.3 over this alternative LO driver HI keeper solution. Combining the new arbiter with a somewhat stronger HI keeper in the predecessor driver might be a good idea too.

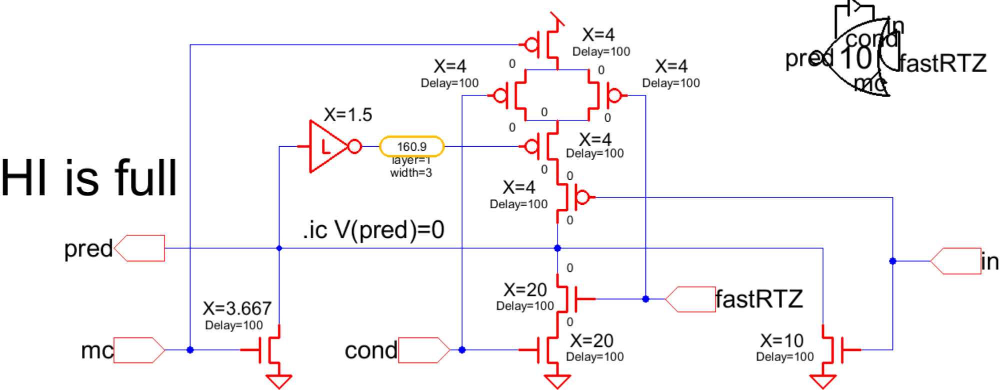


Figure 13: Schematics for the predecessor driver used in the Arbitrated Merge module of **Figure 1**, with its LO driver (NMOS transistors at the bottom) and HI keeper (PMOS transistors at the top). During normal operation either input signal **in** or the logical AND of fast reset signal **fastRTZ** and reset condition **cond**, drive statewire signal **pred** LO. Both these LO drives have a drive size of 10. When signal **pred** is HI and the inputs LO, the keeper maintains the HI voltage level on the statewire until one of the LO-driving input combinations kick in. Usually, a small HI keeper PMOS stack of combined size 1, as we have here, suffices to maintain a HI voltage level; not so when the keeper must also compensate for significant voltage drops induced by contested arbitration.

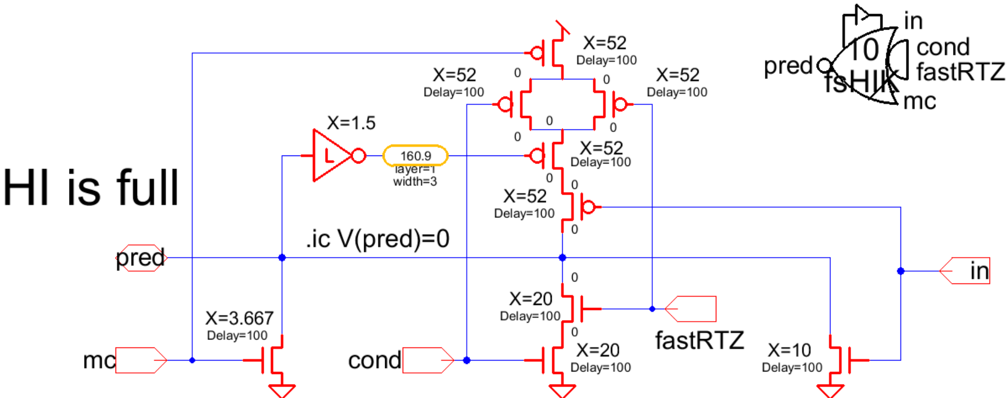


Figure 14: Schematics for the new predecessor driver for the Arbitrated Merge module of **Figure 1**. We changed the drive strengths of the keeper transistors in the PMOS stack so the stack has a total drive of 13. The keeper stack uses 4 PMOS transistors in series. For 4 PMOS gates in series to offer a combined drive size of about 13, each PMOS transistor must have a size four times that value, which is (13)*4=52.

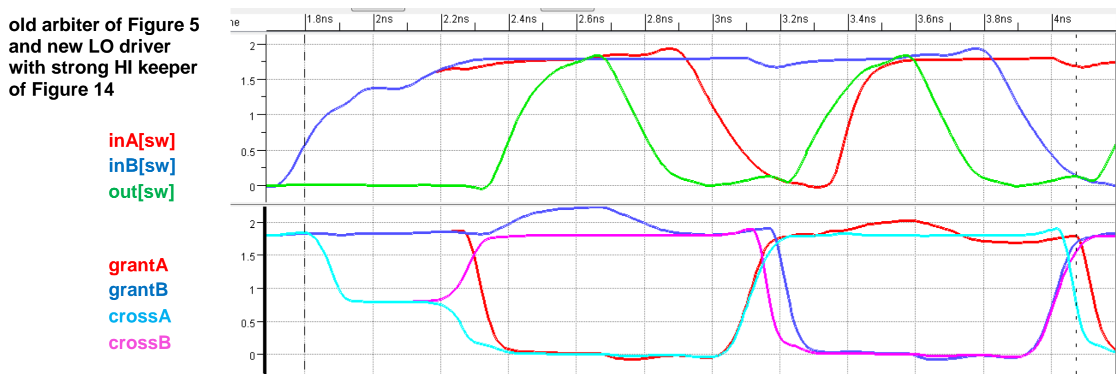


Figure 15: SPICE-level simulation waveforms with the old arbiter implementation of **Figure 5** and the new predecessor driver implementation with a strong HI keeper of **Figure 14**. The HI voltage for inA[sw] respectively inB[sw] drops from 1.8 Volt to somewhere between 1.70 and 1.75 Volt, whenever grantA respectively grantB goes LO. This drop is low enough to not cause issues with noise. Note that each drop is restored almost immediately to 1.8 Volt. The simulation visible cost for improving the voltage levels for inA[sw] and inB[sw] is a reduction in cycle time. The new cycle time for out[sw] is about 900 picoseconds - 17% slower than in **Figure 2**(top) with the old arbiter and the original driver of **Figure 13**.

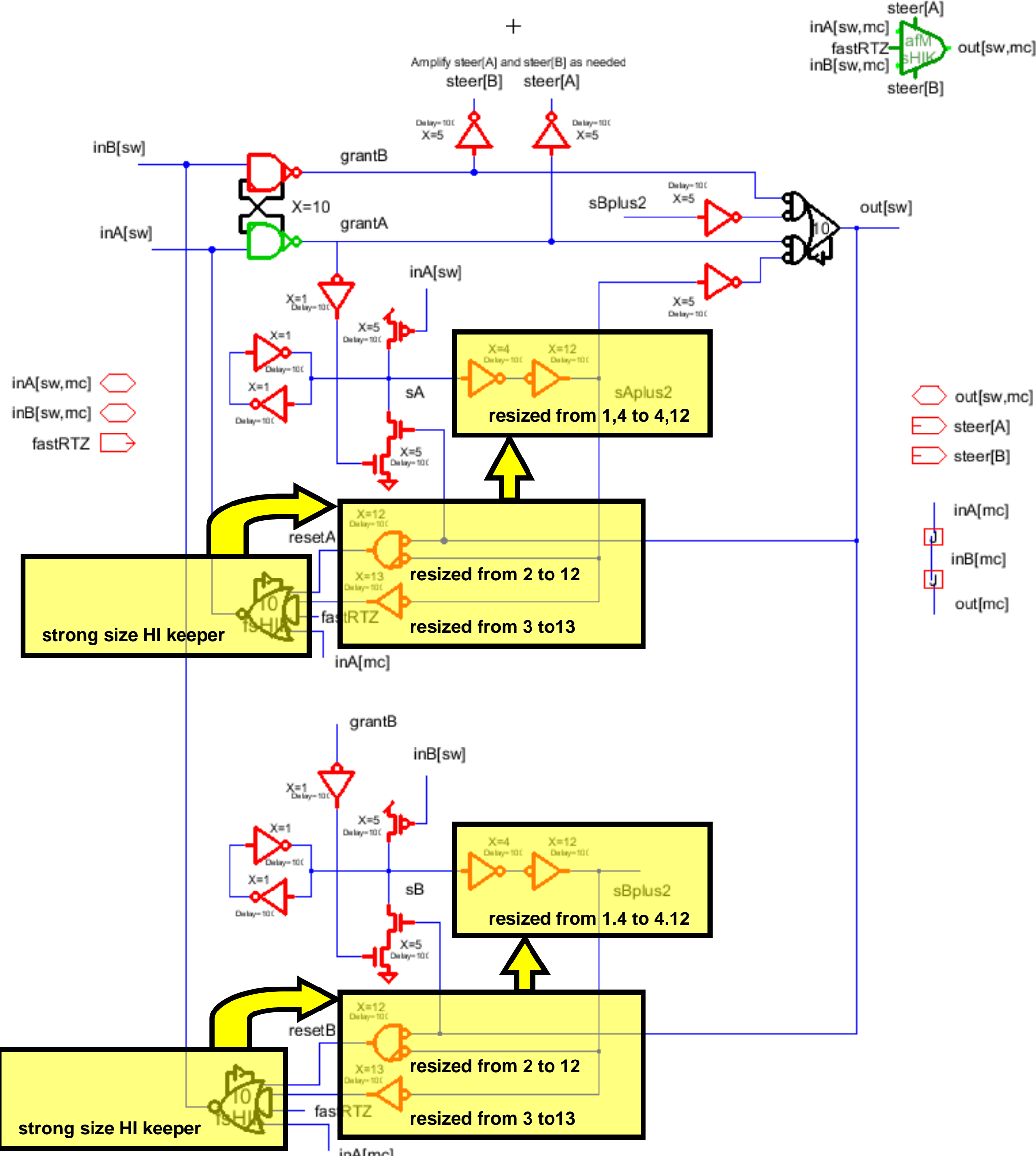


Figure 16: Arbitrated Merge with the new predecessor drivers of **Figure 14**. The new predecessor drivers require a step-up-of-3 driving gate of size ((10 * (1 / 3)) + (52 * (2 / 3)) / 3 = 12.7 for input **resetA** respectively **resetB,** which we rounded down to size 12. The drive strength for the inverter of the fast reset condition input to each predecessor driver must be around ((20 * (1 / 3)) + (52 * (2 / 3)) / 3 = 13.8, which we rounded down to 13. The original gate size for the inverted input AND gates that generate **resetB** and **resetA** was 2, as can be seen in **Figure 1**. The original gate size for the inverters generating the fast reset condition was 3. These increased gate sizes force a size increase in the preceding two upstream inverters in series, i.e., the inverters from **sA** to **sAplus2** respectively **sB** to **sBplus2**. These inverters must be upsized from their original sizes 1 and 4 in **Figure 1** to sizes 4 and 12. The resize dependencies are marked by yellow arrows and boxes. The larger size gates result in an overall larger capacitance for the Arbitrated Merge module with the new predecessor driver with the strong HI keeper. The larger capacitance causes the 20% higher cycle time for **out[sw]** in **Figure 15**.

5. Gate Re-ordering Solutions

Our current Telescope GasP modules are designed for small forward and backward latencies of 2 gate delays each. This low-latency design criterion leaves no room for re-ordering the gates in the forward path. The path necessarily goes through the arbiter first and through the successor driver next. As a result, the statewires go straight into the arbiter, and experience its full input capacitance. To help the statewires cope better with the input capacitance of the arbiter, we re-implemented the arbiter and we re-implemented the predecessor driver. That is what Sections 3 and 4 were about.

Here, in Section 5, we take a look at modules that have a larger forward latency, and thus leave room for re-ordering the gates in the forward path. Examples are Click modules and traditional 6-4 GasP modules. **Figure 17** shows a traditional 6-4 GasP implementation for the Arbitrated Merge module. It is called gaspDemandMerge, and is described in [13]. Its forward path goes through 6 gates, and starts with an arbiter. The two fire signals act as steering signals for the data multiplexers and ensure that the data of the granted statewire are forwarded to the successor module.

The larger forward latency puts less stress on the module by allowing more time and gates for amplification. Note that each arbiter output in **Figure 17** drives only one gate of size 5. Compare this to the arbiter in the Telescope GasP module of **Figure 1**, where each output drives three gates of sizes respectively 1, 5, 10. As a result, the 6-4 GasP module uses a lighter-weight arbiter, of size 5, than the Telescope GasP version, which uses an arbiter of size 10. The flattened transistor schematics for this lighter-weight arbiter follows in **Figure 18**. Topology and design approach are the same as for the size 10 arbiter in **Figure 5** used in the Telescope GasP version. The SPICE-simulated waveforms in **Figure 19** show that the gaspDemandMerge suffers from equally severe voltage drop issues on the statewires coming into the arbiter as do the original Arbitrated Merge waveforms in **Figure 2**(top).

We seek to lower this voltage drop to acceptable levels, by re-ordering the gates in the forward path so that (1) the forward path starts with two inverters, and (2) the resulting forward path goes through 6 gates, and (3) the arbiter remains unchanged.

Figure 20 shows a re-implementation for the gaspDemandMerge, where the gates are re-ordered as specified above, and the arbiter remains unchanged. The corresponding SPICE-level simulated waveforms follow in **Figure 21**. The cycle time for both implementations, with and without re-ordering, is 650 picoseconds. So, by re-ordering the gates, we did not jeopardize the cycle time. We do, however, lose two step-up-of-3 amplifications for generating the fire signals. The fire signals must be amplified outside of the module, as needed. This loss of amplification is the price we pay for obtaining the perfect waveforms shown in **Figure 21**, without the earlier voltage drops on the statewires of the incoming handshake channels.

Note:
We could delete the third requirement “(3) the arbiter remains unchanged”, and make the arbiter part of the step-up-of-3 logical effort sizing and amplification strategy. For the gaspDemandMerge module, we could keep the arbiter either in its present form between the two inverters and the inverted input AND gates or in its inverted form between the first inverter and an inverter followed by the inverted input AND gate per arbiter output. By sizing these first four gates on each forward path appropriately we can maintain the original amplification for the fire signals. If possible, however, we would prefer to not change the arbiter.

gaspDemandMerge

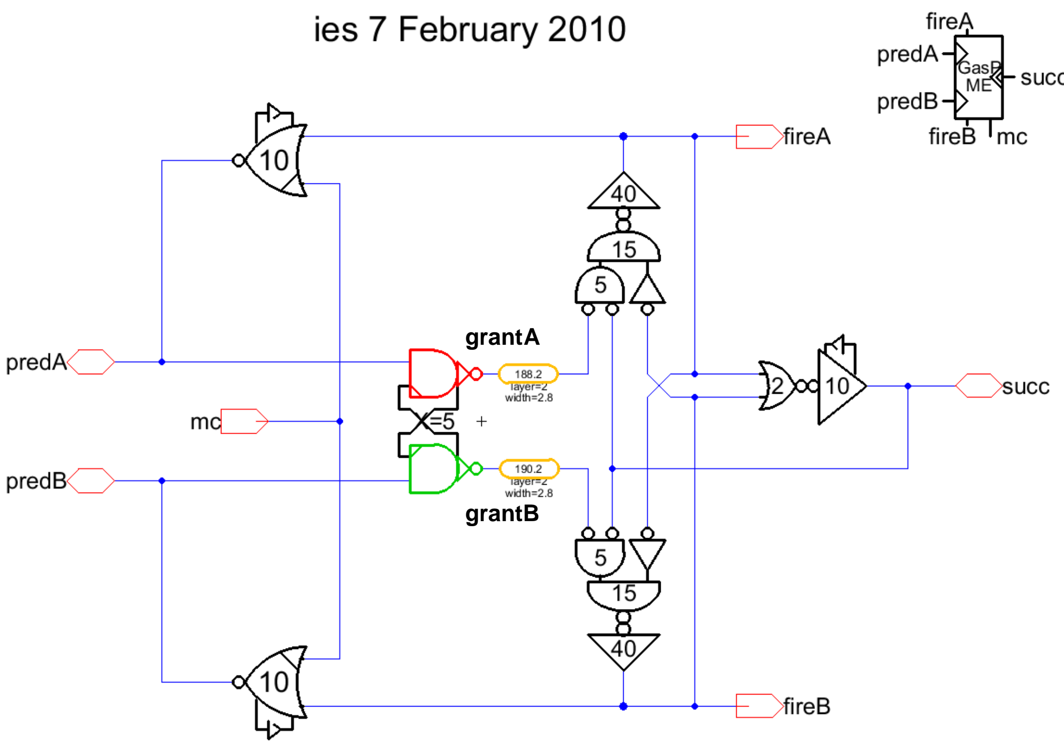


Figure 17: 6-4 GasP equivalent of the Arbitrated Merge module in **Figure 1**. The 6-4 GasP version is called **gaspDemandMerge** [13]. Just like the Telescope GasP implementation in **Figure 1**, it has two statewires for incoming channels, called **predA** and **predB** here, a master clear signal, **mc**, and one statewire for the outgoing channel, called **succ** here. Signal **fireA** and **fireB** act as steering signals and local clock signal. Also, just like in **Figure 1**, the statewires for the incoming channels go directly into the arbiter.

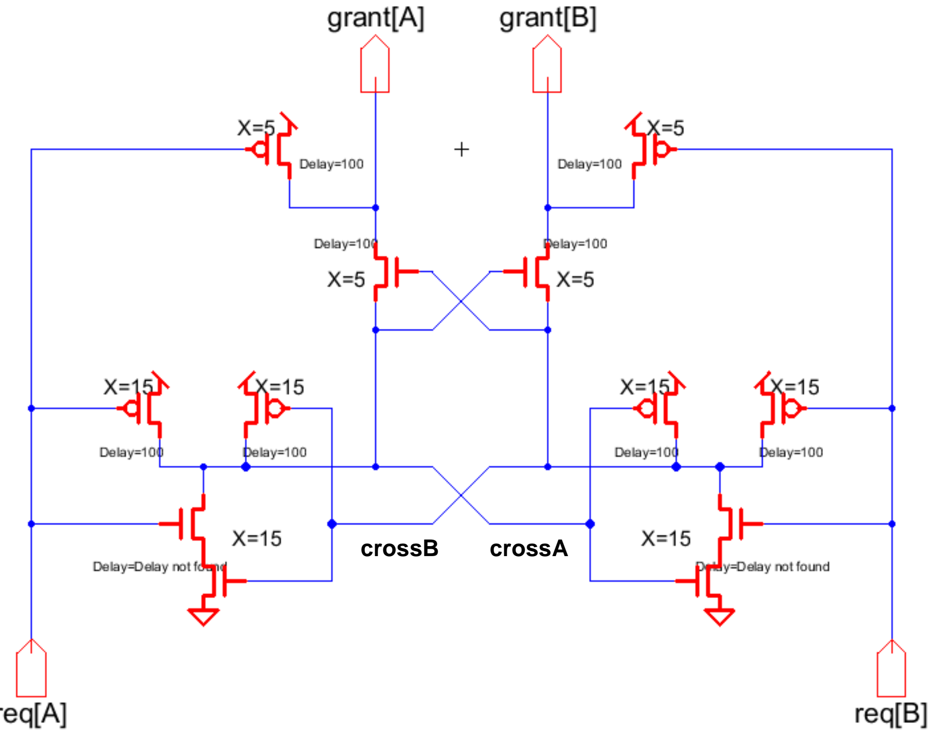


Figure 18: Flattened transistor schematics for the light-weight arbiter of size 5 used in the gaspDemandMerge module in **Figure 17**. Notice the similarity in topology and design style to the larger size 10 version in **Figure 5**.

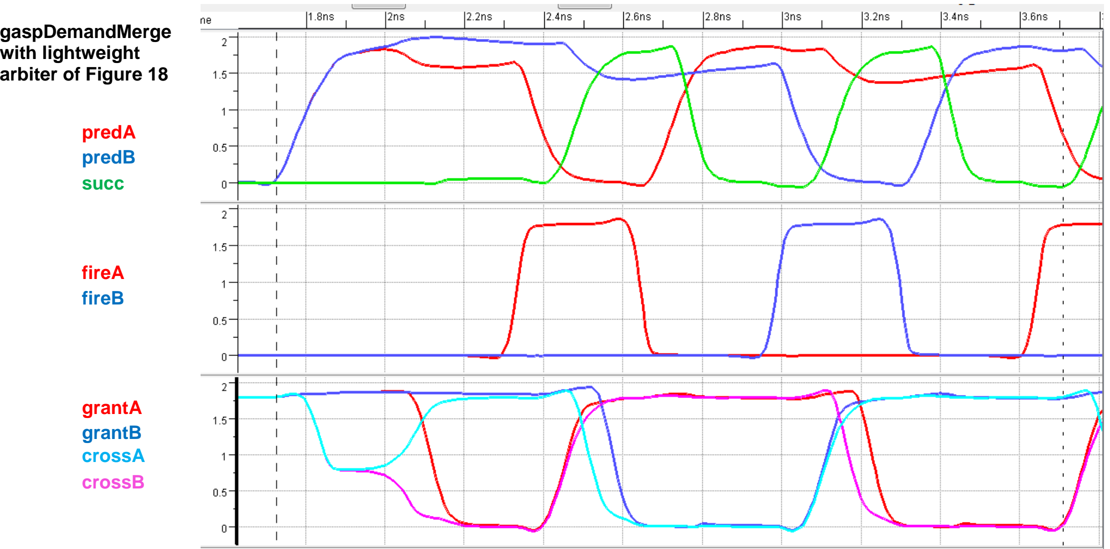


Figure 19: SPICE-level simulation of the gaspDemandMerge module in 6-4 GasP of **Figure 17**, showing the same voltage drop issues on the incoming handshake channels as seen in **Figure 2**(top) for the Telescope GasP version with the old arbiter.

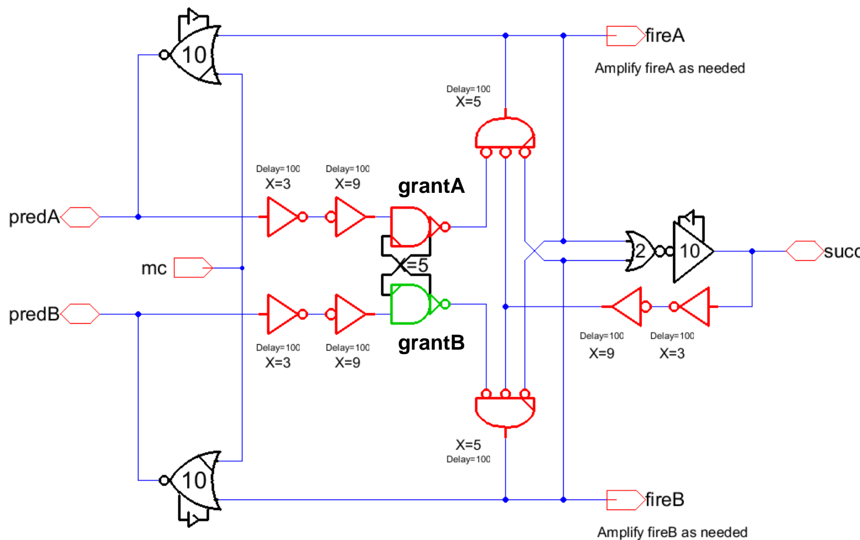


Figure 20: Re-design of the 6-4 GasP gaspDemandMerge module of **Figure 17**, with re-ordered gates in the forward paths and with exactly the same arbiter as before. The two inverters between statewires predA and predB and the arbiter shield the statewires from the large input capacitance of the arbiter and from arbitration noise in the arbiter. Because the same arbiter is used, we lose two step-up-of-3 gates of amplification in the first four gates that drive **fireA** and **fireB**. The reduced drive for **fireA** and **fireB** is good for driving the remaining two serial gates on the forward path to **succ**, and it is good for driving the predecessor drivers on the backward paths to **predA** and **predB**. But the reduced drive is inadequate for steering and clocking the data latches. To do that, **fireA** and **fireB** must be amplified outside the module, as needed.

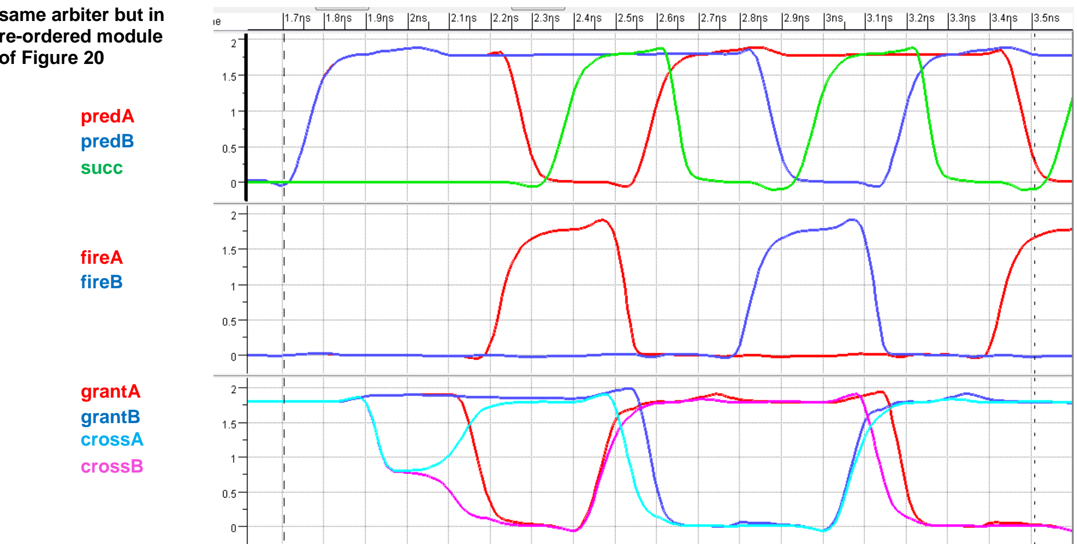


Figure 21: Perfect waveforms for the new gaspDemandMerge module in **Figure 20**, after re-ordering the gates in the forward path to shield the two incoming handshake channels from the arbiter. The new waveforms for predA and predB lack the voltage drops seen in the original predA and predB waveforms in **Figure 19**, which is good.

6. Conclusion and Future Work

In SPICE level simulations of Telescope GasP modules with arbitration [6], we observed voltage drops from 1.8 Volt down to 1.4 Volt, i.e., down to 75% of the supply voltage. The voltage drop occurs on the input that is granted access. The arbiter inputs happen also to function as the statewires of incoming handshake channels. Statewires are bidirectional and provide the communication infrastructure in Telescope GasP and GasP. Consequently, these wires cross longer distances than the wires inside a module. A 25% voltage drop on a statewire makes the communication more susceptible than we like to coupling and noise.

In this document, we investigate various solutions for reducing the voltage drop on arbitrated statewires. The most promising and most general solution comes in the form of a new arbiter implementation with reduced input capacity and reduced sensitivity to arbitration noise. With the new arbiter implementation, the voltage on a granted input drops down to somewhere between 95% and 97% of the supply voltage, which is good enough for most if not all communication scenarios. The new arbiter can be used in any asynchronous circuit in need of arbitration.

In addition to the new arbiter, we can use somewhat stronger HI keepers on arbitrated statewires. We showed that a full-strength HI keeper is not a good solution. A full-strength HI keeper requires a significant layout area and a noticeable upsizing of upstream logic, resulting in an overall higher module capacitance, higher input capacitance during contested arbitrations, and lower cycle time. But combining the new arbiter with a somewhat stronger HI keeper on the arbitrated statewires may be a good idea to pursue. We leave this for future work.

For circuit families with longer forward latencies, such as 6-4 GasP and Click, we can re-order or rather re-organize the gates in the forward path and place low capacitance gates, e.g. two low capacitance inverters in series, between the statewires of the incoming handshake channels and the inputs of the arbiter. The low capacitance gates will shield the statewires from the high capacitance arbiter. This solution can also be used with the new arbiter, in case the statewires cannot endure any voltage drop.

The disadvantage of gate-reordering is a potential loss in amplification in the forward path, unless we are willing to include the arbiter in the amplification effort by increasing its drive size appropriately. In this document, we chose to re-use the same arbiter and not upsize it when we reordered the 6-4 GasP implementation of the gaspDemandMerge. The re-ordered gaspDemandMerge loses two step-up-of-3 amplifications in the steering and local clock generation compared to the original gaspDemandMerge implementation, but has adequate drive strength to drive both the successor and the predecessor statewires. Circuit families with forward paths of 8 gates or more, like for instance an 8-2 GasP implementations of the gaspDemandMerge, may well have adequate drive strength after gate re-ordering to drive the successor and predecessor statewires as well as the steering and local clock logic. Also, control-data bundling constraints that require more than 2 extra gate delays in the control path would fit well at arbitration points: the extra 2 gates help set off the loss in amplification in a re-ordered arbiter path. Let's keep this in mind for near-future work on control-data delay matching in Telescope GasP and GasP.

Appendix H

Arbiter Design: Noise Robustness

Reprinted with permission from the Asynchronous Research Center (ARC) at Portland State University. This internal ARC report serves as appendix to Chapter 7 and can be found as reference [36] of this thesis. The full citation is as follows:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland.
  Further Arbiter Design Improvements: Noise Robustness. *Technical Report, ARC2014-smg01, Asynchronous Research Center, Portland State University*, February, 2014.

ASYNCHRONOUS RESEARCH CENTER
Portland State University

Subject: **Further Arbiter Design Improvements: Noise Robustness**
Date: **February 25, 2014**
From: **Swetha Mettala Gilla and Marly Roncken**
ARC#: **2014-smg01**

References:

[1] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Arbiter Design Improvements for GasP, *ARC2012-smg05, Technical Report, Asynchronous Research Center, Portland State University, 2012.*
Note: ARC2012-smg05 is included as Appendix G in Swetha's Ph.D. thesis.
[2] Charles Seitz. System Timing. Chapter 7 in *Introduction to VLSI Systems, Carver Mead and Lynn Conway (Eds), Addison-Wesley*, pages 218-262, 1980.
[3] Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland. Telescope GasP Narrowcast & Arbitrated or Data Driven Control: Merge, Branch, and Distribute. *ARC2012-smg03, Technical Report, Asynchronous Research Center, Portland State University,* 2012.
Note: ARC2012-smg03 is included as Appendix B in Swetha's Ph.D. thesis.
[4] — GasP Scan Chain Implementation for Init, Go, and Single Step, *ARC2013-smg08, Ibidem.*
Note: ARC2013-smg08 is included as Appendix E in Swetha's Ph.D. thesis.
[5] — Scan Testing GasP using a JTAG Controller, *ARC2013-smg09, Ibidem.*
Note: ARC2013-smg09 is included as Appendix F in Swetha's Ph.D. thesis.
[6] Charles Molanr, Ian Jones, Bill Coates, and Jon Lexau. A FIFO Ring Performance Experiment, *Proc. Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 279-289, 2014.
[7] Steve Rubin, Electric VLSI Design System, *Static Free Software at http://www.staticfreesoft.com/.*

ABSTRACT
This document motivates our extended arbiter design, which has the name "Arbiter D" in Ivan's design library for Electric [7], and compares its performance and behavior against that of our previously preferred arbiter, "Arbiter B". Arbiter implementations can be used in any self-timed circuit. For instance, the Arbitrated Merge module in [3] uses an arbiter. We also use arbiters between the go and the state wire signals in order to properly stop the circuit when it's operating in normal, i.e., self-timed, mode, in a way similar to the Sun Microsystems solution published in [6].

In this report, we'll compare arbiter designs in the context of scan-testable single-track GasP circuits [4,5]. The earlier design of Arbiter B is motivated in ARC reports [1,3]. By comparing its implementation against the original arbiter implementation by Chuck Seitz, called the interlock element in [Figure 7.25, 2], we realized that we were missing part of the Seitz arbiter features. The missing part made our design less robust against noise. We extended the design of Arbiter B to that of Arbiter D to overcome this drawback. This report contains SPICE simulated waveforms for our 180nm CMOS implementations of GasP, showing the presence of noise sensitivity in Arbiter B and confirming the noise robustness of Arbiter D.

Acknowledgements: we gratefully acknowledge Professor Dr. Xiaoyu Song for encouraging and supporting Swetha to conduct this research as part of her PhD thesis work.

This document contains information developed at the Asynchronous Research Center at Portland State University. You may disclose this information to whomever you please. You may reproduce this document for any not-for-profit purpose. Reproduction for sale is strictly forbidden without written consent of the author. Copies of the material must contain this notice.

Table of Contents

1. Introduction 2
2. From input-robust Arbiter B to noise-robust Arbiter D 2
3. Test Setup and Simulation Results 7
4. Conclusion 11

1. Introduction

The Arbiter is a coordination circuit necessary for asynchronous systems. The purpose of the arbiter is to grant only one request at a time. When an arbiter gets two requests at a time, it must decide which request to grant first and to delay the second request until the first has completed its associated actions. We can use arbiters to achieve a single input to single output communication mode of operation between multiple input and output channels. The Telescope GasP implementations for the Arbitrated Merge in [Figures 17-18, 3] use an arbiter circuit to achieve this – see [3] for more details.

Our arbiter designs are based on the interlock element by Chuck Seitz [Figure 7.25, 2]. The Electric library [6] with the arbiters is called GasP. It has four Arbiter circuits:

1) Arbiter A: our original arbiter design.
2) Arbiter B: Arbiter A with low input capacitance.
3) Arbiter C: Arbiter A with extra keeper transistors.
4) Arbiter D: Arbiter B with extra keeper transistors.

Arbiters A and B are explained in [1]. While simulating waveforms of Arbiter A in a Telescope GasP (TGasP) arbitrated Merge design, we discovered non-negligible voltage drops on one of the incoming channels of the Merge [3]. We subsequently attributed these drops to the channel waiting to be granted access by the arbiter. This led to a re-design of the arbiter, Arbiter B that greatly diminishes the voltage drop to a negligible amount – see [1] for details.

In this report, we start with Arbiter B, and show a test scenario with a floating grant signal and its catastrophic effect in the presence of noise interference. We can prevent the floating and thus the noise sensitivity and its catastrophic effect by adding extra keeper transistors on the grant signal, thus yielding Arbiter D.[1] Section 2 explains the designs of Arbiter B and Arbiter D. In Sections 3, we give the test setup and compare the SPICE simulated waveforms of the two designs. Section 4 concludes this document.

2. From input-robust Arbiter B to noise-robust Arbiter D

Our basic arbiter circuit is based on the interlock element designed by Charles Seitz [Figure 7.25, 2], repeated here in Figure 1. The interlock element has input-output pairs Req1-Ack1 and Req2-Ack2. Requests Req1 and Req2 may overlap and even start simultaneously, but acknowledgements Ack1 and Ack2 are strictly mutually exclusive. The circuit in Figure 1 accepts active-low inputs and generates active-low outputs.

[1] Similarly, Arbiter C is the result of adding extra keeper transistors to Arbiter A.

When both input requests Req1 and Req2 for the interlock element are HI, V1 and V2 go LO and the pull-up resistors pull Ack1 and Ack2 HI. When exactly one input request, say Req1, is LO, the circuit grants it by pulling the corresponding acknowledge signal, Ack1, HI. When both Req1 and Req2 are LO, the cross-coupled NOR structure stays in a metastable state until it decides which of the two to grant, in which case it lowers the corresponding acknowledge signal while keeping the other acknowledge signal HI. During metastability, Ack1 and Ack2 are pulled HI by the pull-up resistors.

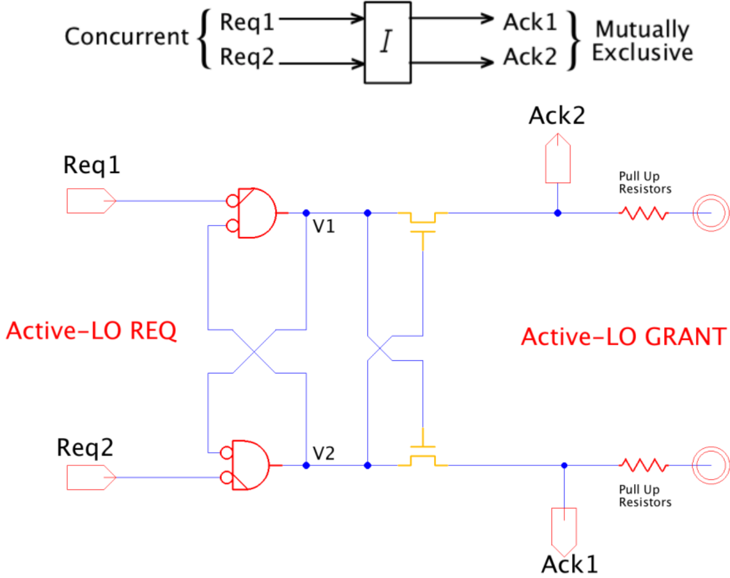


Figure 1: Copy of the *Interlock Element* from [Figure 7.25, 2] by Charles Seitz, without '-tagged extensions for the signal names, as we're not using Seitz his formatting conventions. An alternative name for this circuit is *mutual exclusion circuit* or *mutual exclusion element*. The circuit can be used either alone, or as component of more elaborate elements called arbiters, which is how we use it in GasP and TGasP. It operates via a four-phase handshake per Req-Ack pair, e.g.:

Req1 LO (request), Ack1 LO (granted), Req1 HI (release), Ack1 HI (ready for next request).

The circuit takes active-low requests Req1 and Req2, and produces active-low acknowledgements Ack1 and Ack2. The cross-coupled NOR structure forms a latch that remembers whether the arbiter is taken or ready to grant a new request or still deciding which request to grant. The state in which the cross-coupled NOR structure is still deciding is called "metastable state". The cross-coupled NOR structure can hang for an indeterminate period of time in a metastable state. The pass transistors driven by the cross-coupled NORs will remain inactive until one of the NOR output voltages differs from the other by at least the NMOS transistor threshold voltage, V_{th}, which thus acts as the indicator that the cross-coupled NOR structure is coming out of metastability and has decided which request to grant. Thus, the NMOS transistor structure guarantees that at most one request is granted, i.e., that at most one of the acknowledge signals goes LO. The pull-up resistors for Ack1 and Ack2 keep the acknowledge signal HI during a metastable state and when the corresponding request is not granted.

Our GasP, TGasP and Click circuits currently use the arbiter circuit in Figure 2, which has two input request signals, reqA and reqB, and two active-LO output acknowledge signals, grantA and grantB, just like the Seitz interlock element, but our requests are active-HI and go into cross-coupled NANDs instead of the Seitz cross-coupled NORs. Also, instead of pull-up resistors at the output signals, we use pull-up transistors.

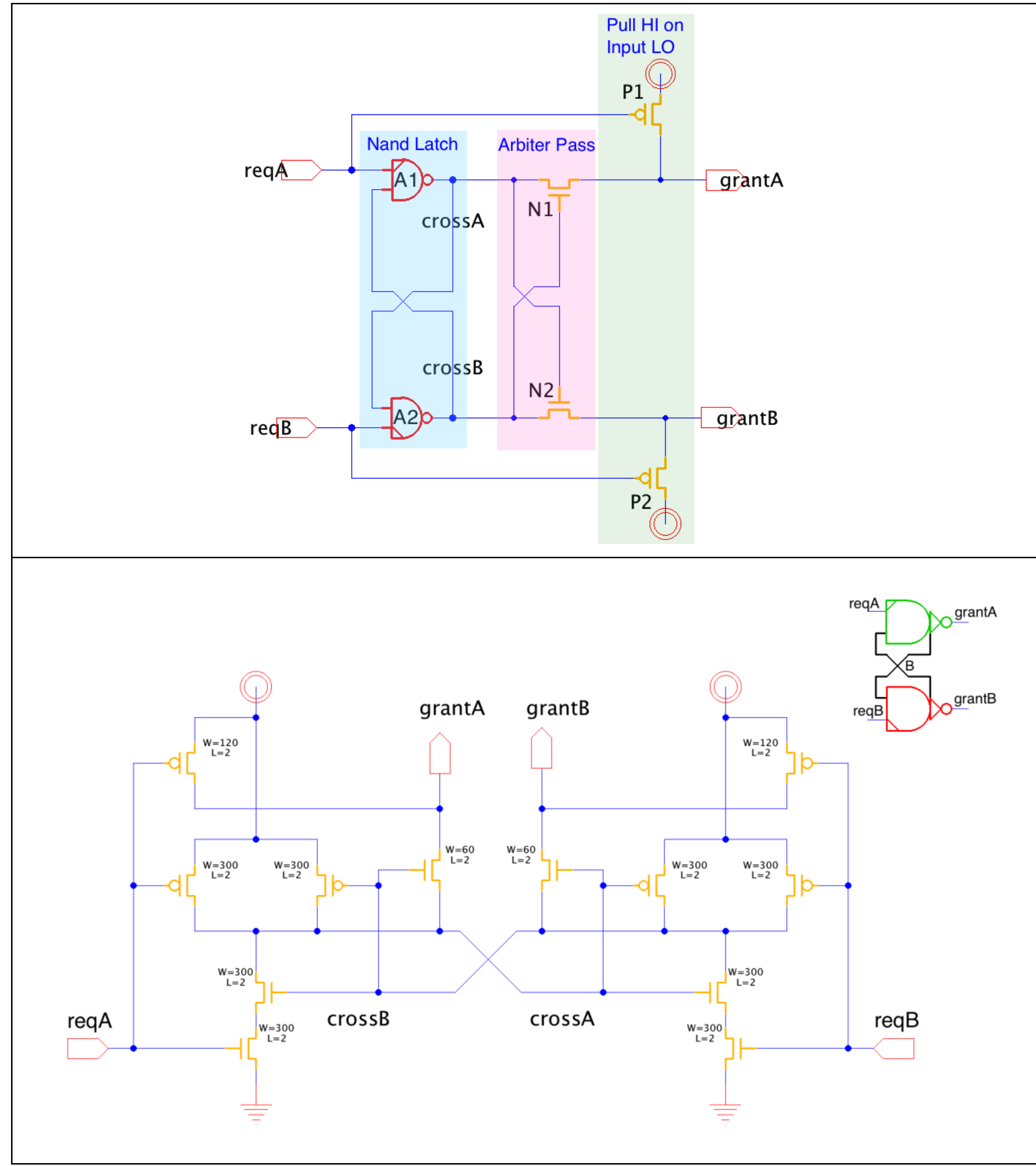


Figure 2: Arbiter B, with cross-coupled NAND structure (top) and as flattened transistor schematics (bottom). Note that the request inputs are connected to the NAND-rail – as also indicated by the triangle in the NAND gate symbol. This is to shield them from the voltage levels and switching activity on crossA and crossB.

It's the pull-up transistors that get us in trouble here, because they don't automatically keep signals grantA and grantB high when the circuit is in a metastable state or when the corresponding request signal is waiting for the arbiter to be released by the other, already granted, request signal.

The signal states for the various input combinations, after metastability resolution, are shown in Table 1. In particular, whenever both inputs are HI, then

- when the arbiter grants or has granted reqA, grantB – initially HI – is left floating.
- when the arbiter grants or has granted reqB, grantA – initially HI – is left floating.

Table 1: Truth table for Arbiter B in Figure 2 (top), after metastability resolution.

Nand Latch inputs		Nand Latch outputs		Arbiter Pass		Pull HI		Arbiter Output	
reqA	reqB	crossA	crossB	N1	N2	P1	P2	grantA	grantB
0	0	1	1	ON	ON	ON	ON	1	1
0	1	1	0	OFF	ON	ON	OFF	1	0
1	0	0	1	ON	OFF	OFF	ON	0	1
1	1	1	0	OFF	ON	OFF	OFF	floating	0
		0	1	ON	OFF	OFF	OFF	0	floating

We overcome this drawback in the new arbiter design, Arbiter D, shown in the Figure 3. Arbiter D has additional HI-keeper transistors, P3 and P4, which prevent grantA and grantB from floating during metastability and during periods where the corresponding request signal is HI and waiting for the arbiter to be released by the other, already granted, request signal, as indicated in Table 2 below.

Table 2: Truth table Arbiter D in Figure 3 (top), after metastability resolution.

Nand Latch inputs		Nand Latch outputs		Arbiter Pass		Pull HI		Keep HI		Arbiter Output	
reqA	reqB	crossA	crossB	N1	N2	P1	P2	P3	P4	grantA	grantB
0	0	1	1	ON	ON	ON	ON	OFF	OFF	1	1
0	1	1	0	OFF	ON	ON	OFF	OFF	OFF	1	0
1	0	0	1	ON	OFF	OFF	ON	OFF	OFF	0	1
1	1	1	0	OFF	ON	OFF	OFF	ON	OFF	1	0
		0	1	ON	OFF	OFF	OFF	OFF	ON	0	1

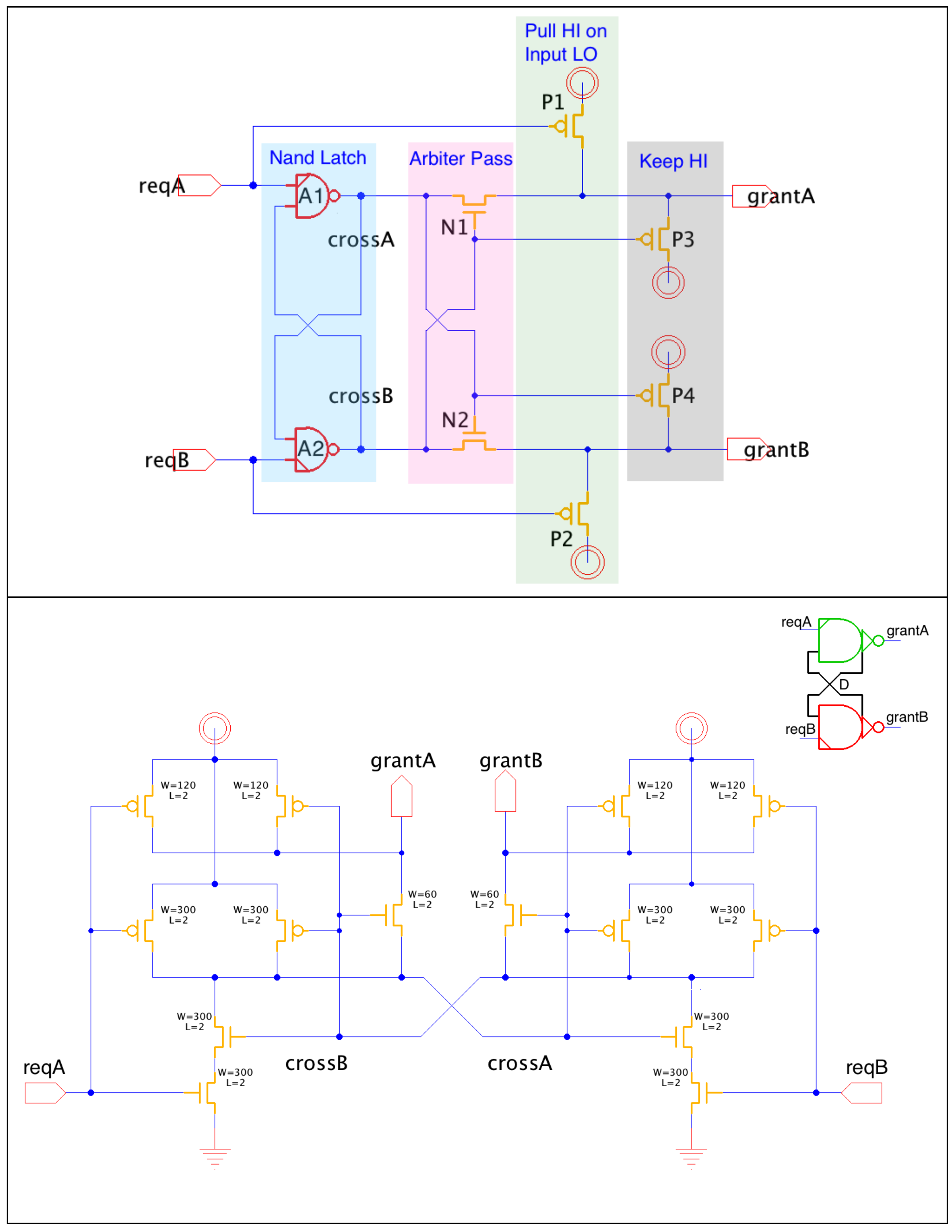


Figure 3: Circuit design of Arbiter D, with cross-coupled NAND structure (top) and in flattened view (bottom),. Arbiter D is an extension of Arbiter B in Figure 2 with additional PMOS keeper transistors to keep grantA and grantB HI during metastability and during periods where the corresponding request signal is HI and waiting for the arbiter to be released by the other, already granted, request signal.

3. Test Setup and Simulation Results

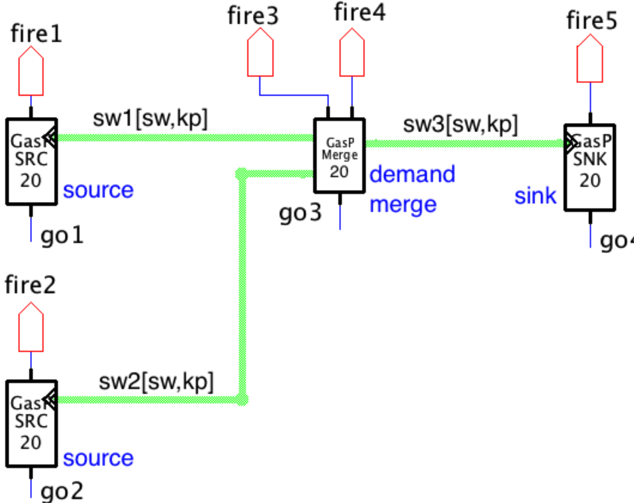


Figure 4: Test setup for the arbiter experiment, not showing the scan chains. The go and sw[kp] signals are connected to the scan chains, as explained in [4]. The test details for initializing a circuit like this one, for single-step test operations and for starting the circuit in normal self-timed mode are explained in [5]. The design of this test setup has been saved in Electric library 180GasP25Feb2014_Electric.

Figure 4 shows the test setup for the arbiter experiments that we use to prove the need for the extra PMOS keeper transistors in Arbiter D. The test configuration merges two data streams coming from GasP Source (SRC) modules into an Arbitrated Merge module that hands the selected streams on to a GasP Sink (SNK) module.

Figure 5 shows a GasP implementation for the Arbitrated Merge, which also goes by the name of Demand Merge. It has two incoming channels, predA and predB, two fire signals, fireA and fireB – one per input channel – and one outgoing channel, succ. The data portion of the channels and the datapath latches are not shown here; only the channel's state wires are, i.e., predA[sw], predB[sw], and succ[sw].

When at least one of the predecessor state wires is HI, indicating the presence of data, and the successor state wire is LO, indicating the presence of space, the Demand Merge arbitrarily grants access to one of the HI predecessor state wires, raises the corresponding fire signal, and then it concurrently (1) raises the successor state wire, (2) lowers the granted predecessor state wire and (3) releases the arbiter.

Given that we use GasP circuits with a cycle time of 10 gate delays, the next request HI on the granted predecessor will take at least 5 gate delays to arrive after the predecessor state wire has gone LO. As a result, the arbiter in the Merge module is fair in the sense that an already waiting request will be granted before the just granted request has time to re-claim the arbiter. The newly granted request will have to wait with firing, though, until succ has completed its handshake, i.e., until succ[sw] has gone LO.

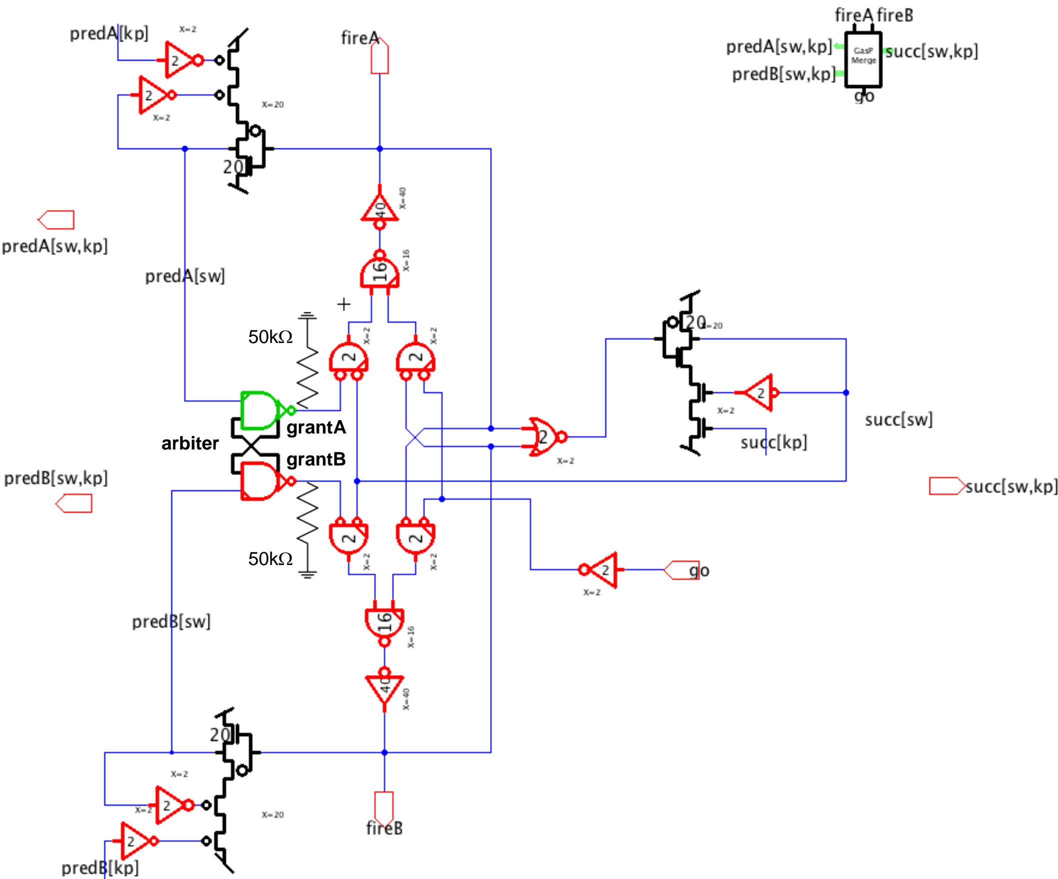


Figure 5: GasP implementation for an Arbitrated Merge Module with two input statewires, predA[sw] and predB[sw], and one output statewire, succ[sw]. Signals predA[kp], predB[kp], succ[kp] and go are the test control signals coming from the scan chains. For this report, we'll use this design with either an Arbiter B or Arbiter D instance. We added the two 50k Ohm resistors on grantA and grantB to mimic noise interference on the grant signal; the resistors are not part of the arbiter or Merge design.

To test the robustness of the arbiter against noise interference, we issue overlapping input requests to the Arbitrated Merge, and use a single-step test approach to arrive in a noise-sensitive state configuration from where we can distinguish and expose the difference in behavior for the noise-sensitive Arbitrated Merge with Arbiter B versus the noise-robust version with Arbiter D. The corresponding lower level test details are programmable from the scan chains. The test goes as follows:

- *Step 1:*
  - **Scan mode**
    Set all go signals LO via the scan chain, then initialize sw1[sw] and sw2[sw] HI and sw[3] LO via the scan chain, then make all go's HI except for go4 of the sink module. Reports [4,5] show how to program the scan chain to write go and sw signals. The circuit is now in partially self-timed mode.

- **Self-timed mode**
  The final go settings will result in a state for which:

  1. The test configuration in Figure 4 performs half a handshake on sw3, the output channel of the Merge. This means that sw3[sw] will go HI and stay HI.

  2. The test circuit performs one and a half handshake on the first granted input channel of the Demand Merge, say sw2 connected to Merge input predB. This means that we'll see predB[sw] go HI, then LO, then HI again. But by the time it goes HI again, the arbiter is already claimed by sw1. In the Merge version with Arbiter B, this will leave grantB floating at a HI voltage level.

  3. The test circuit performs half a handshake on the other input channel, say sw1 connected to Merge input predA. This channel gets granted after the first channel communication releases the arbiter, leaving grantA driven LO. Further progress is stalled at the size-16 NAND gate in the Arbitrated Merge until the Merge successor channel has space again to receive the new input data from sw1, i.e., until sw3[sw] goes LO.

  The resulting state stabilizes in one of two possible end states, one for the Merge with Arbiter B where signal grantB is left floating, and one for the version with Arbiter D where none of the grant signals are left floating. We will wait for the HI voltage level on Arbiter B's grantB to leak away through the 50k Ohm resistor – which we use to mimic noise interference – and to turn into a LO voltage level before we move to test step 2.

- *Step 2*:
  - **Scan mode**
    Make all go signals LO via the scan chain to freeze the circuit, then make both go3 and go4 HI to set the circuit again in partially self-timed mode.

    Note that we change the go signals in two phases: all LO, then selectively HI. We do this to prevents races. Had we changed the go signals all at once and, for instance, go1 and go2 were to go LO much later than go3 and go4 went HI, the Merge module might use the old HI values on go1 and go2 to start new handshakes on sw1, sw2, and sw3 before the new LO values on go1 and go2 would put an end to the creation of new handshakes. By setting all go signals LO first, we avoid races between old and new go settings and the circuit behaviors these enable

  - **Self-timed mode**
    The previous go settings will result in incorrect behavior for the configuration in Figure 4 when the Merge uses Arbiter B. They will yield correct behavior when the Merge uses Arbiter D. With Arbiter B, the fire signals overlap for the first firing action in Step 2. With Arbiter D they remain mutually exclusive.
    The simulation waveforms follow in Figure 6 and Figure 7.

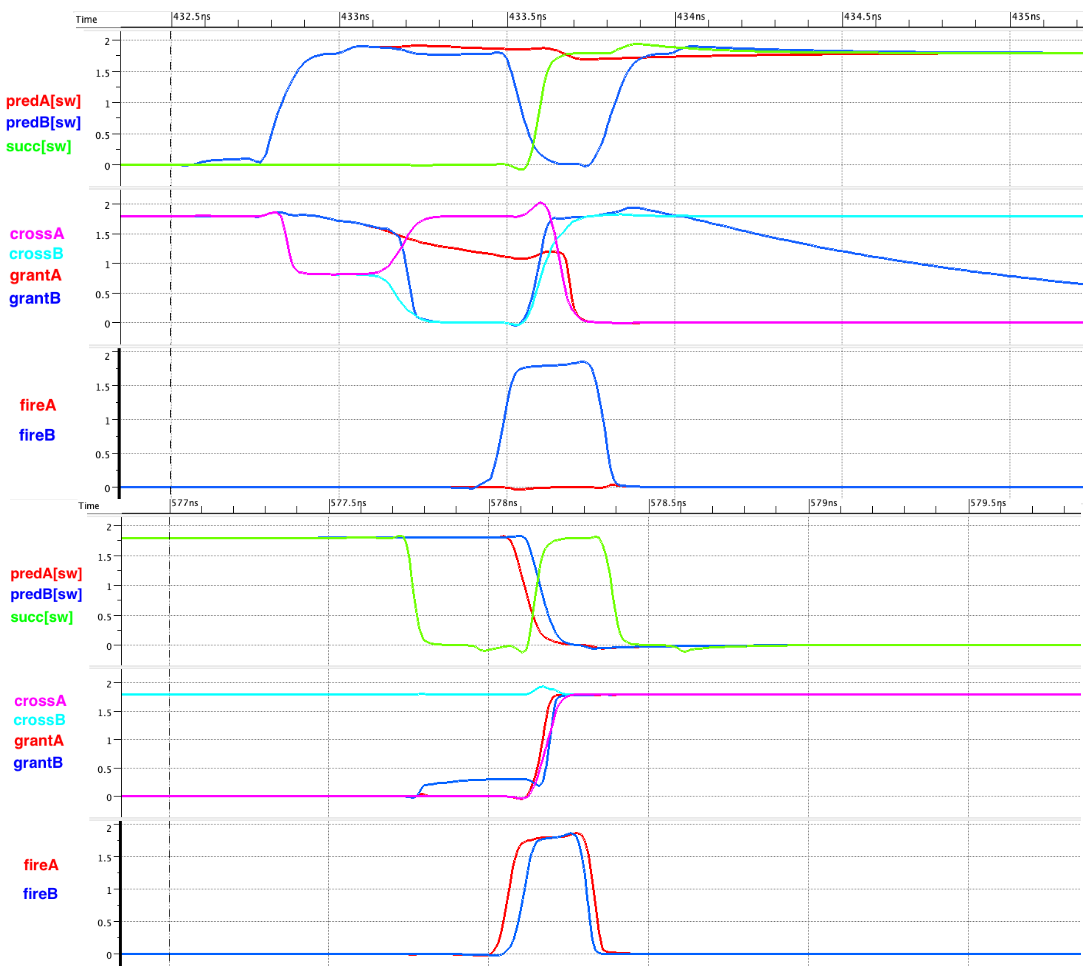


Figure 6: SPICE waveforms for the test configuration and Merge in Figure 4 and Figure 5 using Arbiter B, showing test simulation details for Step 1 (top simulation window) and Step 2 (bottom simulation window), as seen from the viewpoint of the Arbitrated Merge module. Each simulation window shows three sets of waveforms, which are correlated by color.

- **(Step 1 – top)** Signals **predA[sw]** and **predB[sw]** go HI simultaneously in the top row, which results in a metastability period for the arbiter in the middle row that ends with **grantB** going LO, i.e., **predB** granted, which results in **fireB** going HI, which sets in motion three simultaneous events, **succ[sw]** going HI, **predB[sw]** going LO, and **grantB** going HI. The latter event releases the arbiter, which is immediately re-claimed by **predA**, as indicated by **grantA** going LO. Meanwhile **fireB** goes LO and **predB** issues a new handshake as indicated by **predB[sw]** going HI.

  There are two strange behaviors in the top simulation window that hint at noise-sensitivity: the voltage level of **grantA** leaks away during the first grant period (when grantB is LO), and the voltage level of **grantB** leaks away during the second grant period (when grantA is LO). The first period is short enough to maintain the voltage level to not jeopardize the operation, but the second grant period is sufficiently long for the voltage level to switch and create a catastrophic behavior in Step 2 of the test operation.
- **(Step 2 – bottom)** Step 2 of the test starts with **grantA** and **grantB** both at a low voltage level, thus suggesting that both incoming state wires have been granted. Therefore – as soon as **succ[sw]** goes LO – both **fireA** and **fireB** go HI, convoluting the data from **predA** and **predB** into a single **succ** handshake, which is against the mutual exclusion protocol of the Arbitrated Merge.

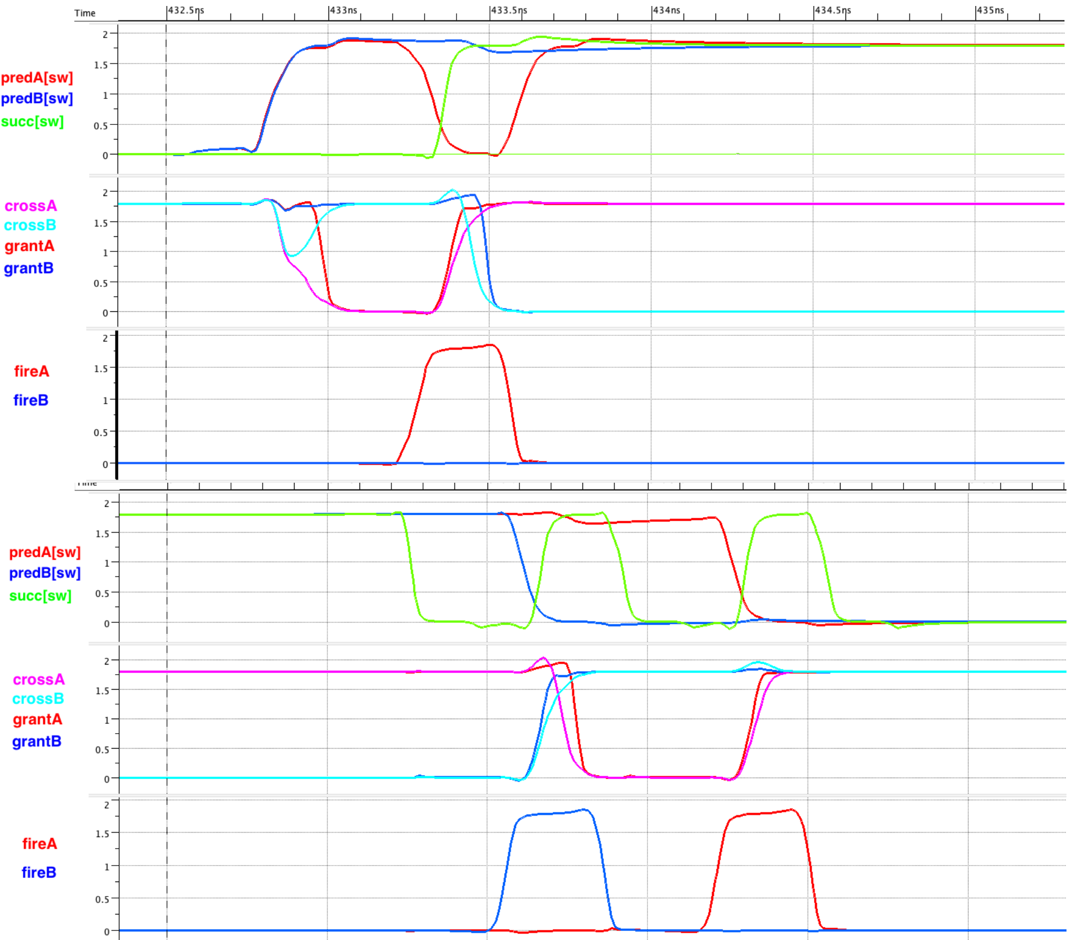


Figure 7: SPICE waveforms for the test configuration and Merge in Figure 4 and Figure 5 using Arbiter D.

- **(Step 1 – top)** This is like the simulation in Figure 6, Step 1, but without the noise-sensitivity indications. Note that the arbiter reverses the ordering of the grants over the ordering shown in Figure 6 by granting **predA** first, i.e., making **grantA** LO, and then granting **predB**, i.e., making **grantB** LO next.
- **(Step 2 – bottom)** Step 2 of the test starts with **grantA** HI and **grantB** LO, thus correctly suggesting that only incoming state wire **predB** is granted communication access. Therefore, as soon as **succ[sw]** goes LO, only **fireB** goes HI. Step 2 shows two correct mutually exclusive handshakes on **succ** – streaming data to **succ** first from **predB** and then from **predA**.

4. Conclusion

In this report, we compared our design for Arbiter B, motivated in ARC reports [1,3], against the original arbiter implementation by Chuck Seitz in [Figure 7.25, 2], and indicated that we're missing part of the Seitz arbiter features. The missing part makes our design less robust against noise. We extended the design of Arbiter B to that of Arbiter D to overcome this drawback. We set up a test simulation environment and presented SPICE simulated waveforms that expose the noise sensitivity in Arbiter B and confirm the noise robustness of Arbiter D.

Appendix I

Bonus: PhD Defense Presentation

Silicon Compilation and Test for Dataflow implementations in GasP and Click

Swetha Mettala Gilla

PhD Dissertation Defense
Asynchronous Research Center (ARC)
Maseeh College of Engineering and Computer Science
Portland State University
3 November 2017

Acknowledgements

Advisors
Dr. Xiaoyu Song (ECE)
Drs. Marly Roncken (CS, ARC)

Committee
Dr. Douglas V. Hall (ECE)
Dr. Robert Daasch (ECE)
Dr. Ivan Sutherland (ECE, ARC)
Dr. Mark Jones (CS, Graduate Office Representative)

ARCwelders
Hoon Park
Chris Cowan
Chris Chen
Yong Hei

Publications

Journals:

- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland
  Hoi Polloi Mutual Exclusion, IEEE Transactions on VLSI
  plan to submit by end of 2017.

Conferences and books:

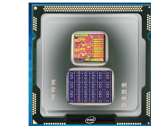
- Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland
  Naturalized Communication and Testing, ASYNC 2015, pages 77-84, 2015.
- Swetha Mettala Gilla
  Testing with MrGO, ASYNC 2015 web site.
- Marly Roncken, Swetha Mettala Gilla, Hoon Park, Robert Daasch, Xiaoyu Song, Chris Cowan, and Ivan Sutherland
  Beyond Carrying Coal to Newcastle: Dual Citizens and Circuits
  Andrey Mokhov (Ed.) This Asynchronous world – essays dedicated to Alex Yakovlev
  Newcastle University, pages 241– 293, July 2016.
- Swetha Mettala Gilla, Marly Roncken, and Ivan Sutherland
  Long Range GasP with Charge Relaxation
  ASYNC 2010, pages 185-195, 2010.
- Swetha Mettala Gilla
  Library Characterization and Static Timing Analysis of Single-Track Circuits in GasP
  M.Sc. Thesis, Electrical and Computer Engineering, Portland State University, October 2010.

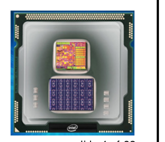
Outline

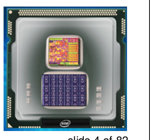
- Overview
- PART I: Silicon compilation for GasP and Click
- INTERMEZZO: New point of view for design and test
- PART II: Test and debug with state and action control
- Contributions

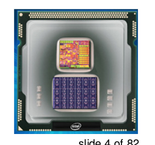
Motivation: why asynchronous?

- Many modern computer systems are distributed over space.
- Examples:
  - **Internet of things** *[Wikipedia]*
    The network of objects or "things" embedded with electronics, software, sensors, and network connectivity, which enables exchange of data.
  - **IBM's TrueNorth**
    Modular chips that act like neurons and form artificial neural networks to run "deep learning algorithms", like Skype's chat translator or Facebook's facial recognition.
  - **Intel's Loihi chip**
    Energy efficient chip that mimics how the brain functions by learning to operate based on feedback from the environment.









PhD Dissertation Defense, Swetha Mettala Gilla — slide 4 of 82

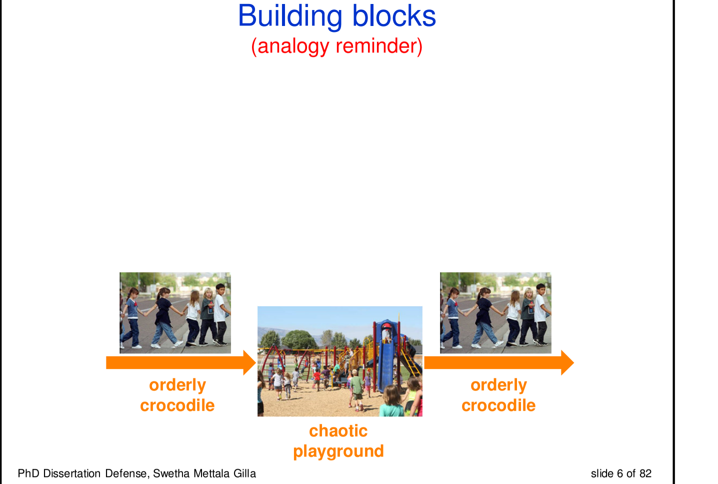
Motivation: why asynchronous?

- We design and study hardware
  - distributed over self-timed components + communication protocols.
- where
  - Inside a component it can be as chaotic as a kindergarten playground which is fine, because components are small enough to control events.
  - Between components, the protocols are as orderly as a "crocodile" which guarantees that the communication is correct.
- This is a scalable system.

orderly crocodile — chaotic playground — orderly crocodile

PhD Dissertation Defense, Swetha Mettala Gilla — slide 5 of 82

Building blocks
(analogy reminder)



PhD Dissertation Defense, Swetha Mettala Gilla — slide 6 of 82

Building blocks

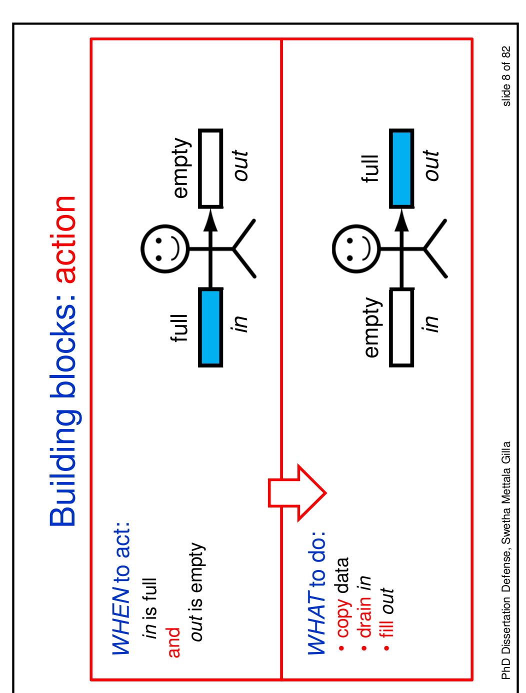
handshake component
module

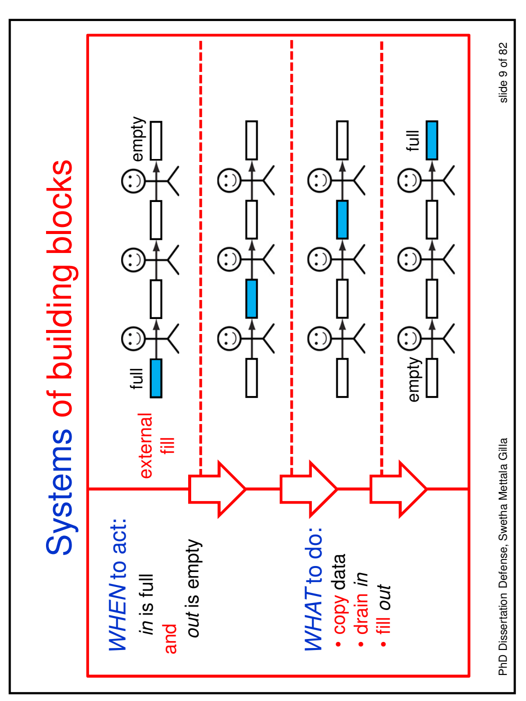
communication channel
handshake channel

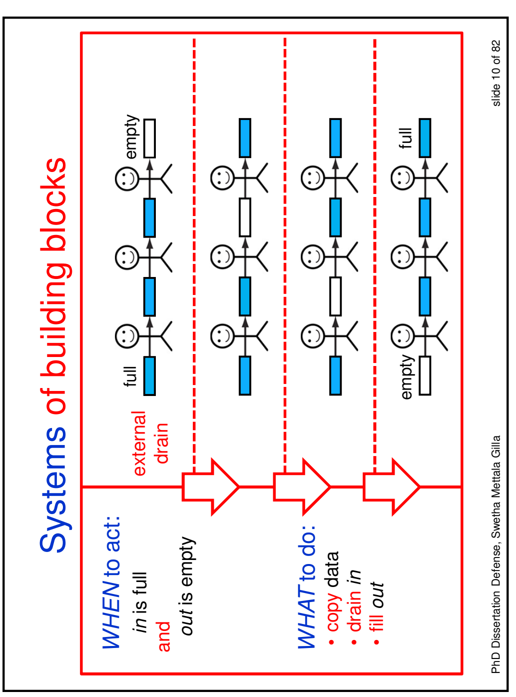
in

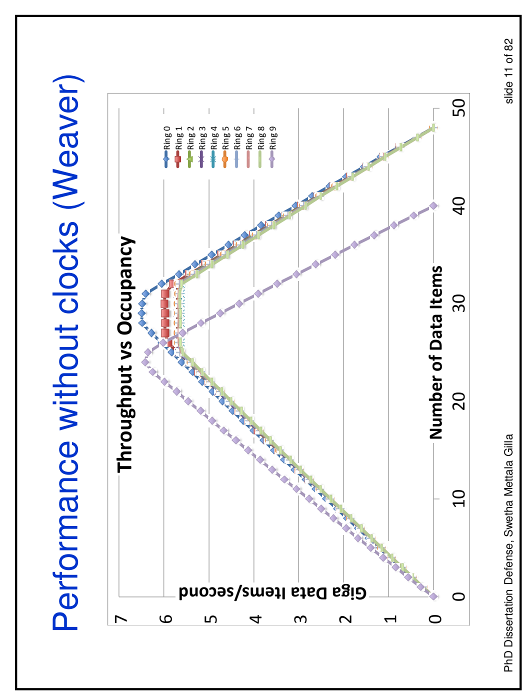
out

PhD Dissertation Defense, Swetha Mettala Gilla — slide 7 of 82


Building blocks: action
WHEN to act:
in is full
and
out is empty
WHAT to do:
• copy data
• drain in
• fill out
full
empty
in
out
empty
full
in
out
PhD Dissertation Defense, Swetha Mettala Gilla
slide 8 of 82


Systems of building blocks
WHEN to act:
in is full
and
out is empty
WHAT to do:
• copy data
• drain in
• fill out
external
fill
full
empty
empty
full
PhD Dissertation Defense, Swetha Mettala Gilla
slide 9 of 82


Systems of building blocks
WHEN to act:
in is full
and
out is empty
WHAT to do:
• copy data
• drain in
• fill out
external
drain
full
empty
empty
full
PhD Dissertation Defense, Swetha Mettala Gilla
slide 10 of 82


Performance without clocks (Weaver)
Throughput vs Occupancy
Giga Data Items/second
Number of Data Items
0
1
2
3
4
5
6
7
0
10
20
30
40
50
PhD Dissertation Defense, Swetha Mettala Gilla
slide 11 of 82

Building blocks at start of thesis

- My building blocks use GasP and Click circuits which differ in handshake protocols and gates.
- GasP circuits are highly customized. They have better logical effort, low power and high speed.
- Click has the most synchronous asynchronous circuits we know. These circuits work better with standard industry tools for static timing analysis, placement and routing.

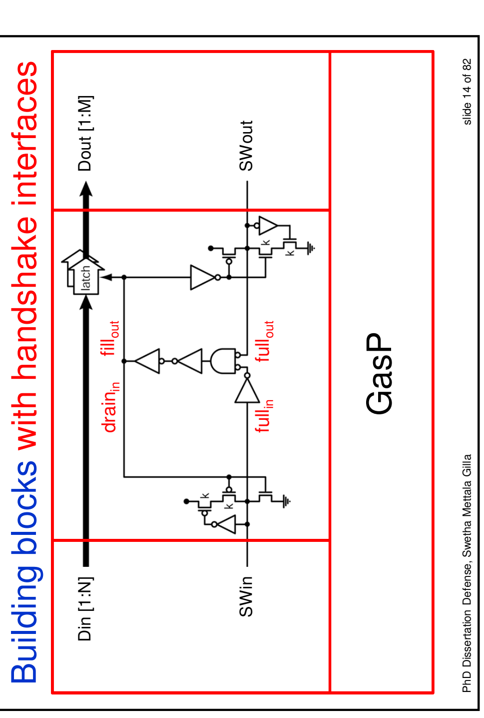
Handshake protocols

- Handshakes *encode* full, empty, and valid data when full
- Examples:
  - 2-phase return-to-zero (RTZ) with bundled data (used in GasP)
    full: statewire is high / empty: statewire is low

    Voltage, bundled data, statewire, empty, valid, full, empty, valid, full, empty, Time
  - 2-phase non-RTZ with bundled data (used in Click)
    full: request ≠ acknowledge / empty: request = acknowledge

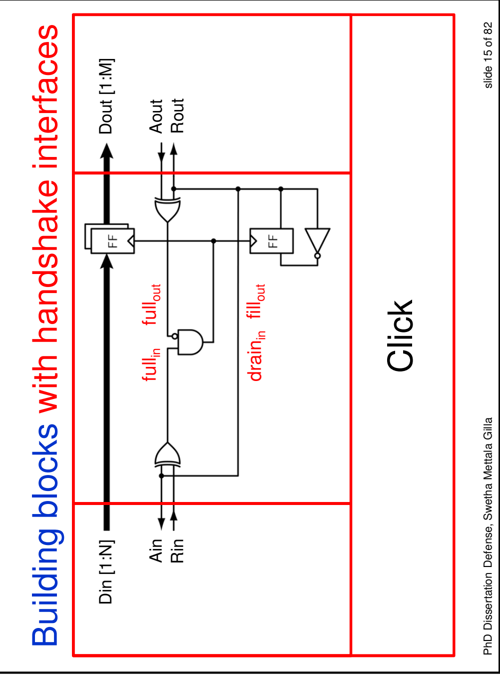
    Voltage, bundled data, request, acknowledge, empty, valid, full, empty, valid, full, empty, Time

Building blocks with handshake interfaces



GasP

Building blocks with handshake interfaces



Click

State-of-the-ARC at the start of my thesis

- For design, we used:
  - Electric schematic editor to build circuits in GasP.
  - ARCwelder compiler to build circuits in Click.
- For test, we knew that:
  - Sun-Oracle Labs used a partial scan and functional test to characterize the performance of GasP circuits.
  - Philips used full scan and structural test ("one-shot test") to detect stuck-at faults in Click circuits.

PhD Dissertation Defense, Swetha Mettala Gilla — slide 16 of 82

Focus of my thesis

GOAL:

- Create a general automated design and test solution
- for self-timed distributed systems
- that works with any self-timed circuit family
- including mixed systems with GasP and Click.

PhD Dissertation Defense, Swetha Mettala Gilla — slide 17 of 82

Focus of my thesis – continued

PART I:

GOAL:

- Extend ARCwelder from Click to **Click and GasP.**

APPROACH:

1. Extend GasP style to support Telescope GasP (TGasP) to match all existing Click designs in ARCwelder.
2. Adapt ARCwelder to generate GasP and TGasP.

OBSERVATION:

- Adaptation required **too many** code changes.

PhD Dissertation Defense, Swetha Mettala Gilla — slide 18 of 82

Focus of my thesis – continued

INTERMEZZO – NEW POINT OF VIEW:

PART I influenced how the ARC views design and test

DESIGN:

- Previous : various handshake interfaces
  Now : common full-empty interfaces
- Previous : components have both communication and computation
  Now : separate communication (links) and computation (joints)

TEST:

- Previous : state-centric test control (scan)
  Now : state control (scan) and action control (MrGO, go)

PhD Dissertation Defense, Swetha Mettala Gilla — slide 19 of 82

Focus of my thesis – continued

PART II:

GOAL:

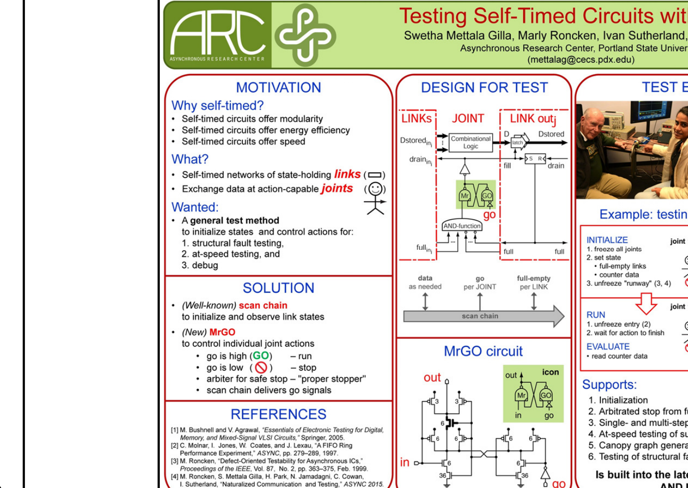
- Work out the new test point of view.

APPROACH:

1. Implement action control: MrGO and go signals.
2. Extend scan test for states to **states and actions.**
3. Create a standard test interface.
4. Demo all of this on real silicon.

OBSERVATION:

- It works perfectly ! (see Poster).



My contributions

System design

- My research influenced ARC's new design and test point of view.
- I created Telescope GasP – a GasP extension still relevant today.
- I extended ARCwelder to support (T)GasP.

Arbitrated circuits

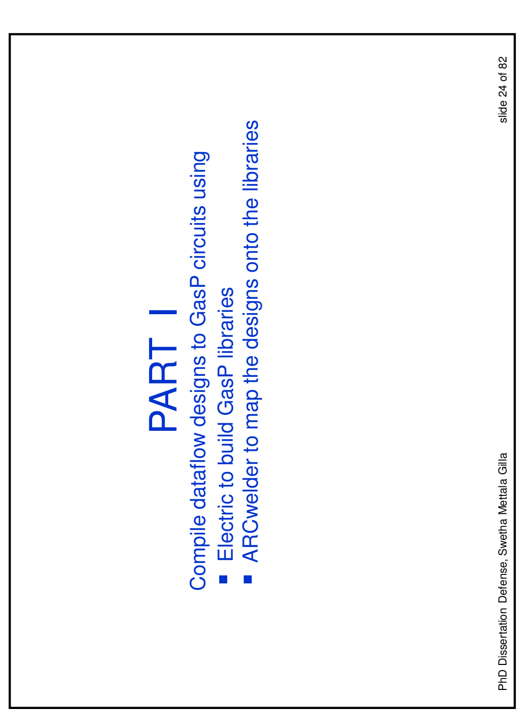
- With MrGO, arbiters are everywhere now.
- I improved the noise tolerance for arbiter inputs and outputs.
- I created a mathematical foundation to size arbiters for speed.

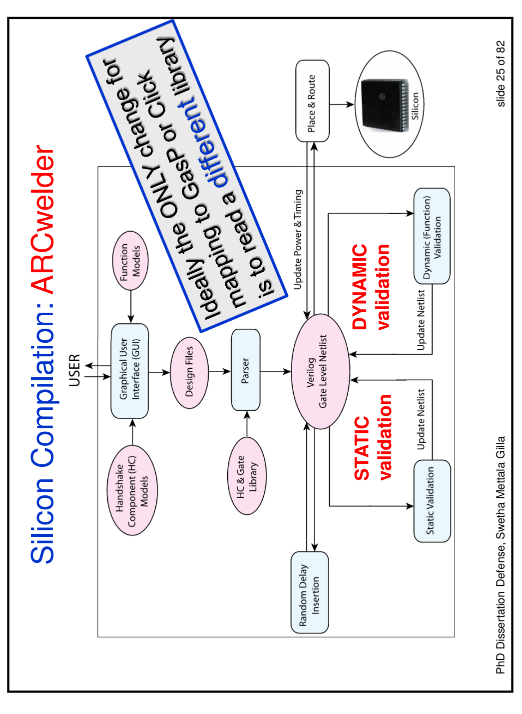
Test, debug, and performance characterization

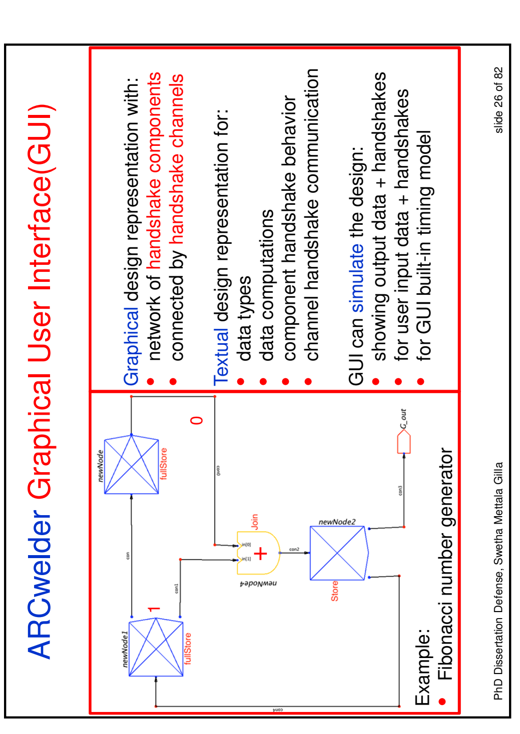
- I implemented MrGO and scan.
- I built and demo-ed the combined MrGO-scan solution on silicon.
- I proposed an initialization solution which works at power-up.

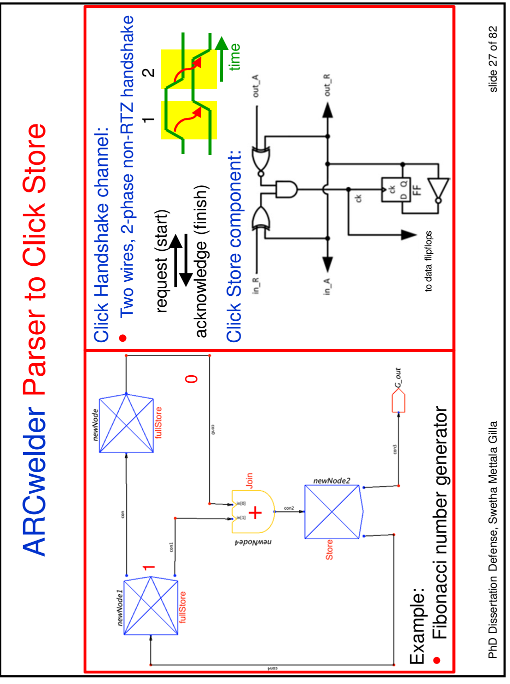
Outline

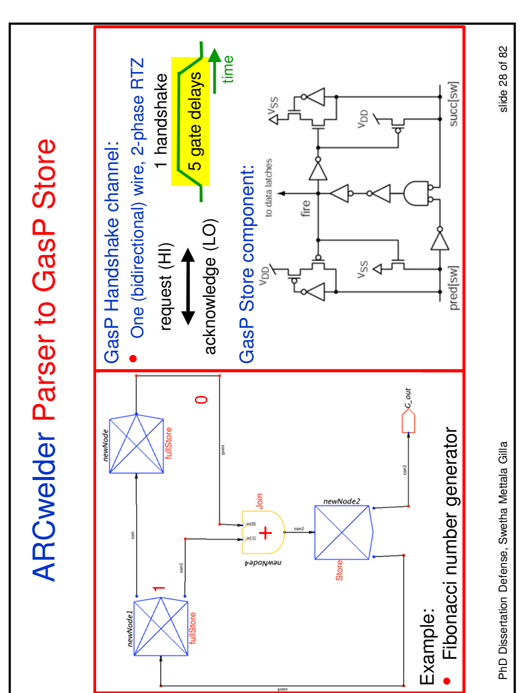
- Overview
- PART I: Silicon compilation for GasP and Click
- INTERMEZZO: New point of view for design and test
- PART II: Test and debug with state and action control
- Contributions

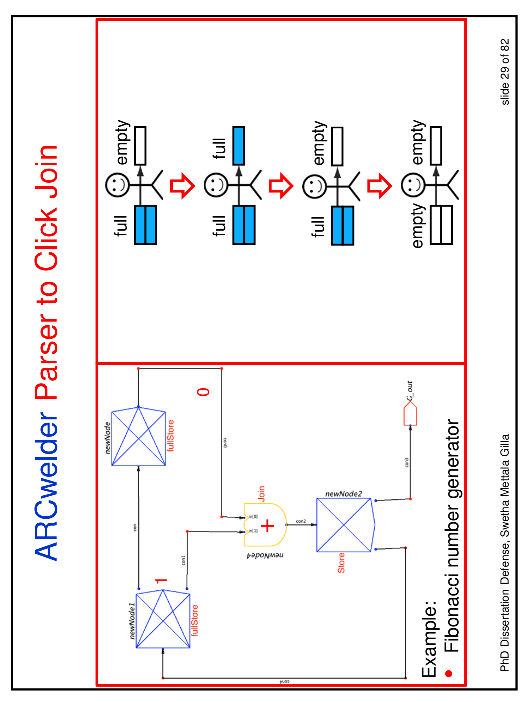

PART I
Compile dataflow designs to GasP circuits using
Electric to build GasP libraries
ARCwelder to map the designs onto the libraries
PhD Dissertation Defense, Swetha Mettala Gilla
slide 24 of 82

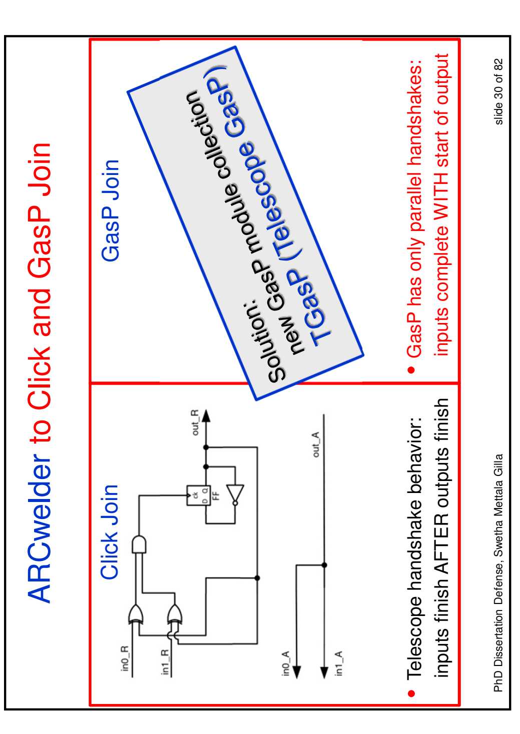

Silicon Compilation: ARCwelder
USER
Handshake Component (HC) Models
Graphical User Interface (GUI)
Function Models
Design Files
HC & Gate Library
Parser
Verilog Gate Level Netlist
Random Delay Insertion
Static Validation
Update Netlist
STATIC validation
DYNAMIC validation
Dynamic (Function) Validation
Update Power & Timing
Place & Route
Silicon
Ideally the ONLY change for mapping to GasP or Click is to read a different library
PhD Dissertation Defense, Swetha Mettala Gilla
slide 25 of 82

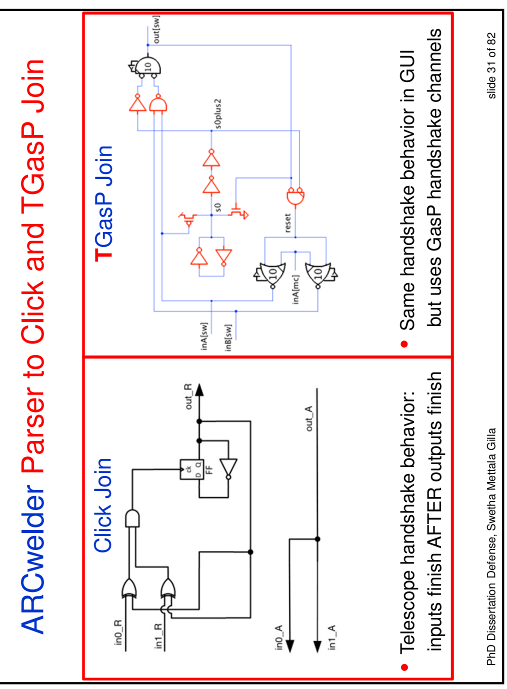

ARCwelder Graphical User Interface(GUI)
newNode1
newNode
fullStore
1
0
Join
newNode4
newNode2
Store
C_out
Example:
Fibonacci number generator
Graphical design representation with:
network of handshake components
connected by handshake channels
Textual design representation for:
data types
data computations
component handshake behavior
channel handshake communication
GUI can simulate the design:
showing output data + handshakes
for user input data + handshakes
for GUI built-in timing model
PhD Dissertation Defense, Swetha Mettala Gilla
slide 26 of 82


ARCwelder Parser to Click Store
newNode1
newNode
fullStore
1
0
Join
newNode4
newNode2
Store
C_out
Example:
Fibonacci number generator
Click Handshake channel:
Two wires, 2-phase non-RTZ handshake
request (start)
acknowledge (finish)
1
2
time
Click Store component:
in_R
out_A
in_A
out_R
ck
FF
to data flipflops
PhD Dissertation Defense, Swetha Mettala Gilla
slide 27 of 82

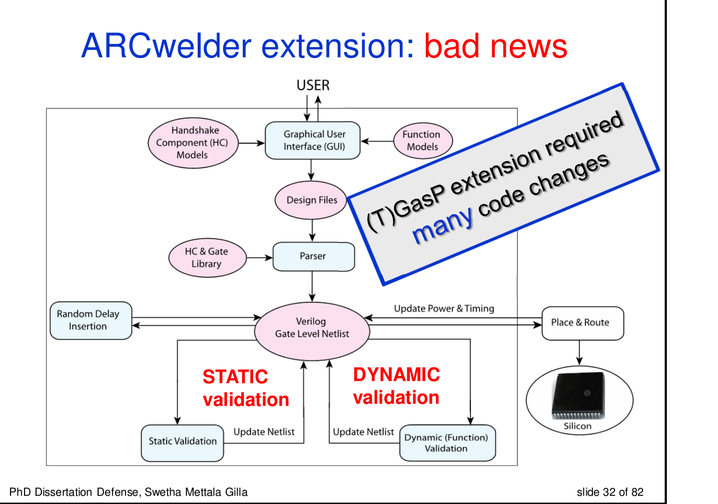

ARCwelder Parser to GasP Store
newNode1
fullStore
1
newNode
fullStore
0
Join
newNode4
newNode2
Store
C_out
Example:
• Fibonacci number generator
• GasP Handshake channel:
One (bidirectional) wire, 2-phase RTZ
request (HI)
acknowledge (LO)
1 handshake
5 gate delays
time
GasP Store component:
to data latches
fire
pred[sw]
succ[sw]
PhD Dissertation Defense, Swetha Mettala Gilla
slide 28 of 82


ARCwelder Parser to Click Join
newNode1
fullStore
1
newNode
fullStore
0
Join
newNode4
newNode2
Store
C_out
Example:
• Fibonacci number generator
full
empty
full
full
full
empty
empty
empty
PhD Dissertation Defense, Swetha Mettala Gilla
slide 29 of 82


ARCwelder to Click and GasP Join
Click Join
in0_R
in1_R
out_R
in0_A
in1_A
out_A
FF
GasP Join
Solution:
new GasP module collection
TGasP (Telescope GasP)
• Telescope handshake behavior:
inputs finish AFTER outputs finish
• GasP has only parallel handshakes:
inputs complete WITH start of output
PhD Dissertation Defense, Swetha Mettala Gilla
slide 30 of 82


ARCwelder Parser to Click and TGasP Join
Click Join
in0_R
in1_R
out_R
in0_A
in1_A
out_A
FF
TGasP Join
inA[sw]
inB[sw]
inA[mc]
reset
s0
s0plus2
out[sw]
• Telescope handshake behavior:
inputs finish AFTER outputs finish
• Same handshake behavior in GUI
but uses GasP handshake channels
PhD Dissertation Defense, Swetha Mettala Gilla
slide 31 of 82

ARCwelder extension: bad news



PhD Dissertation Defense, Swetha Mettala Gilla — slide 32 of 82

ARCwelder extension: good news

- New TGasP family
  PRO:
  - GasP + TGasP provide a 1-1 replacement for Click.
  - Previous GUI designs can now be mapped to GasP + TGasP.
  - Telescope protocols provide safe dataflow by design.

  CON:
  - TGasP is slower than GasP.
- ARCwelder extension works for the entire flow including validation
  - Static: combinational GasP loops require special care.
  - Dynamic: custom GasP requires powerful simulator like Modelsim.
- Initial experimental results for TGasP are promising
  - (T)GasP is 24-34% faster than Click for no-delay datapaths.
  - Speeds are more similar when datapaths take more time.

PhD Dissertation Defense, Swetha Mettala Gilla — slide 33 of 82

Outline

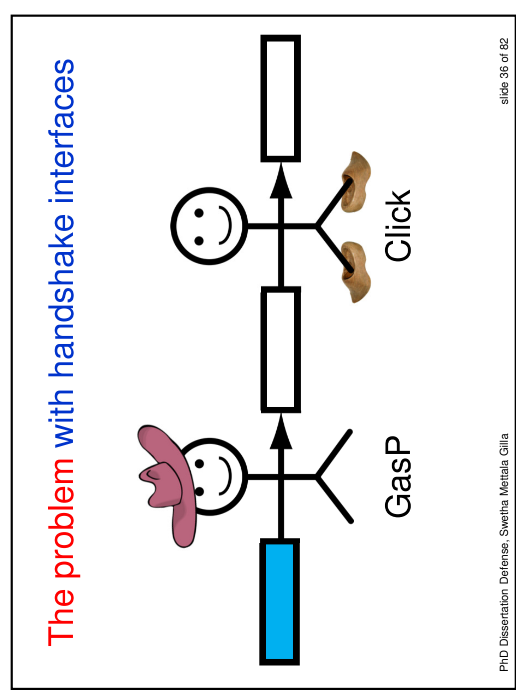
- Overview
- PART I: Silicon compilation for GasP and Click
- INTERMEZZO: New point of view for design and test
- PART II: Test and debug with state and action control
- Contributions

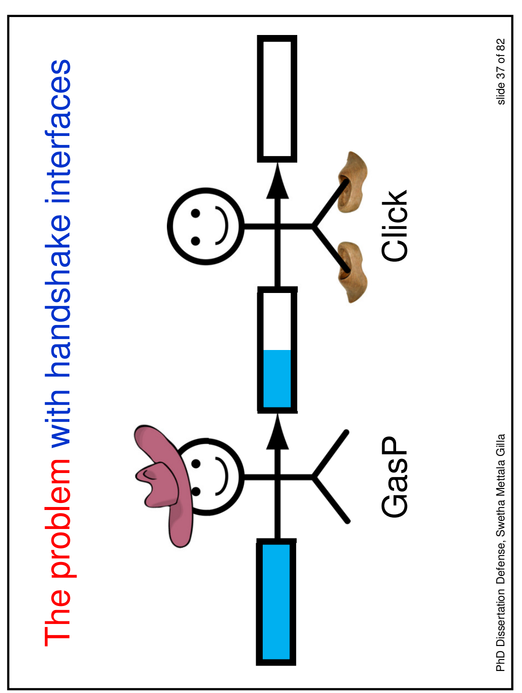
PhD Dissertation Defense, Swetha Mettala Gilla — slide 34 of 82

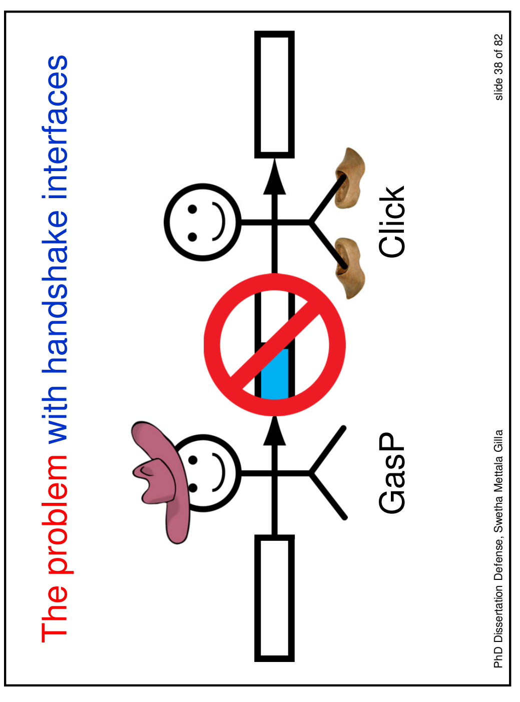
INTERMEZZO

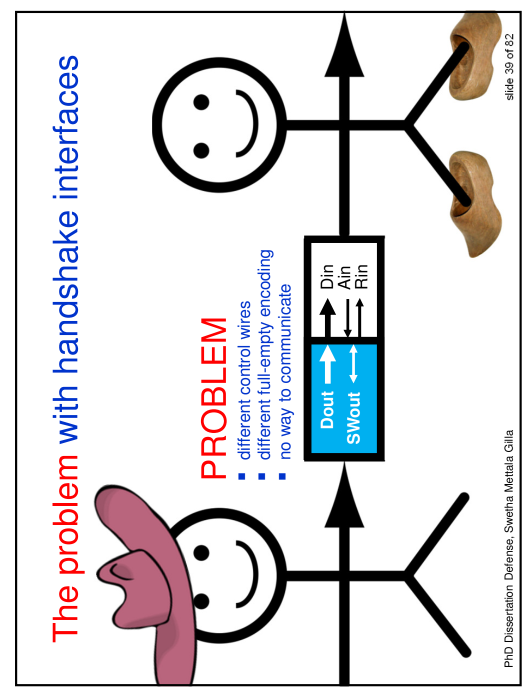
INTERMEZZO I: New point of view for DESIGN
- common full-empty interfaces
- separate communication (links) and computation (joints)

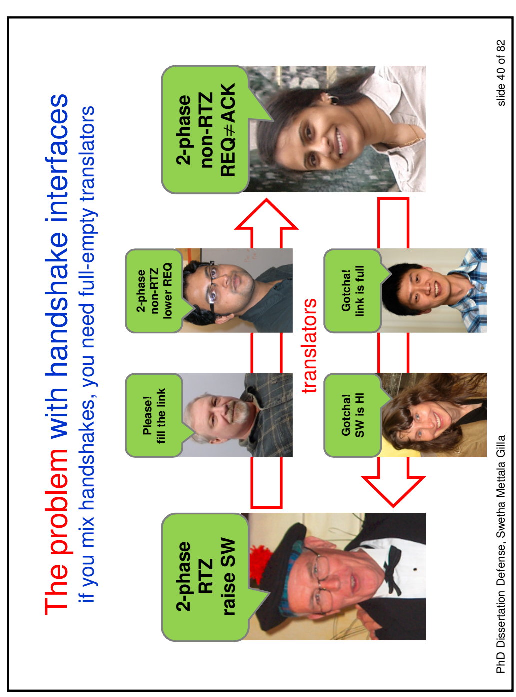
PhD Dissertation Defense, Swetha Mettala Gilla — slide 35 of 82

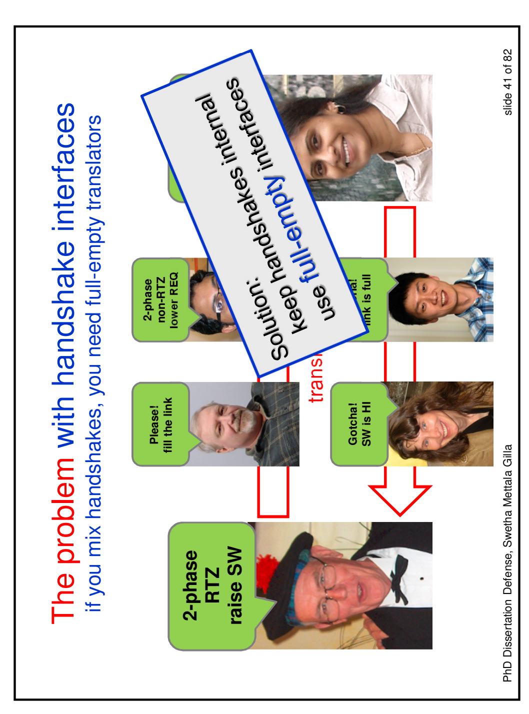

The problem with handshake interfaces
GasP
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 36 of 82

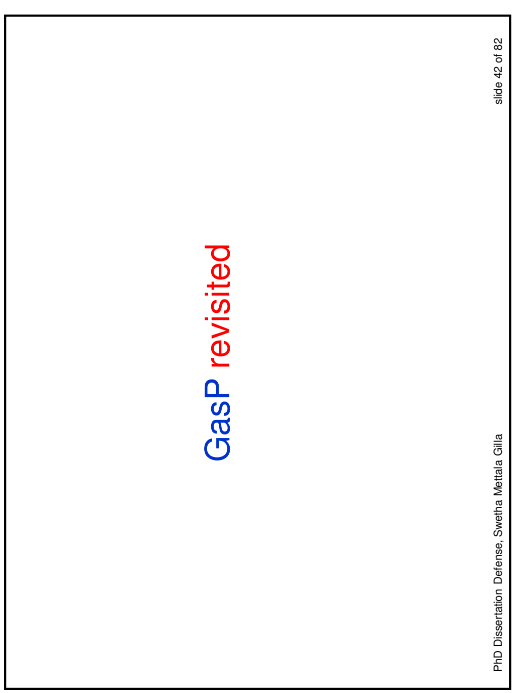

The problem with handshake interfaces
GasP
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 37 of 82

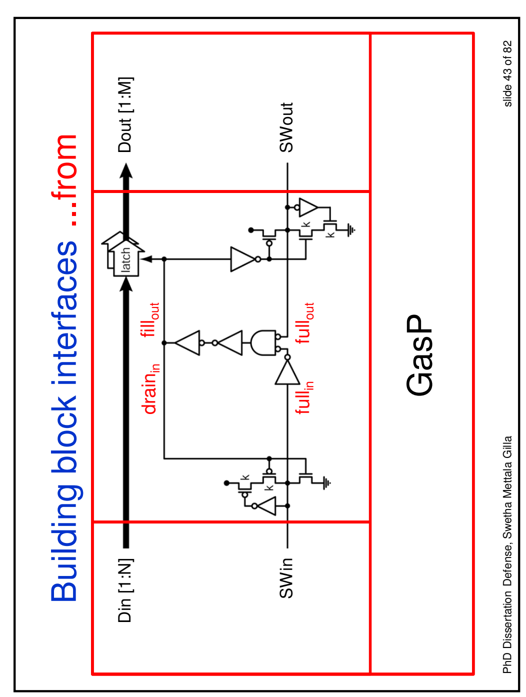

The problem with handshake interfaces
GasP
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 38 of 82

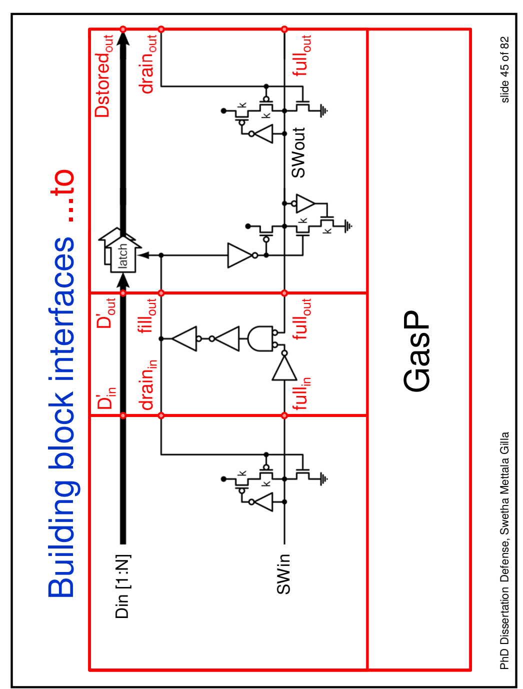

The problem with handshake interfaces
PROBLEM
different control wires
different full-empty encoding
no way to communicate
Dout
SWout
Din
Ain
Rin
PhD Dissertation Defense, Swetha Mettala Gilla
slide 39 of 82

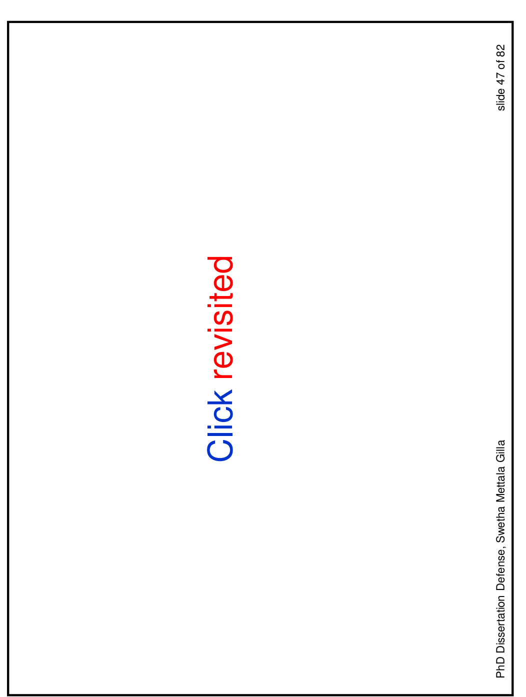

The problem with handshake interfaces
if you mix handshakes, you need full-empty translators
2-phase RTZ raise SW
Please! fill the link
2-phase non-RTZ lower REQ
2-phase non-RTZ REQ≠ACK
translators
Gotcha! SW is HI
Gotcha! link is full
PhD Dissertation Defense, Swetha Mettala Gilla
slide 40 of 82

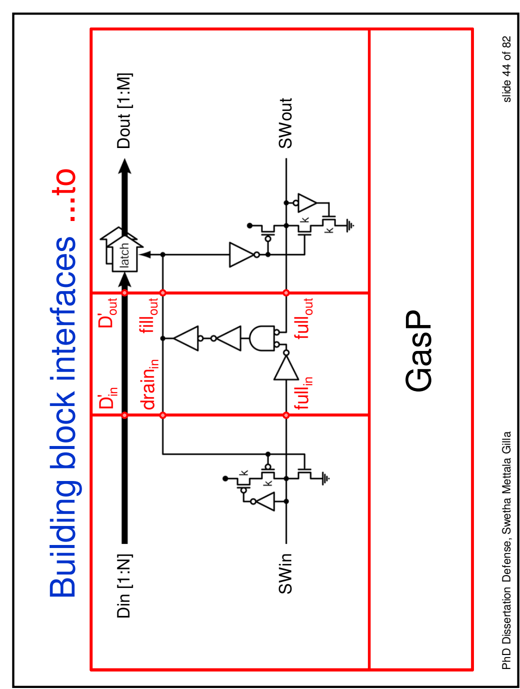

The problem with handshake interfaces
if you mix handshakes, you need full-empty translators
2-phase RTZ raise SW
Please! fill the link
2-phase non-RTZ lower REQ
Gotcha! SW is HI
Solution:
keep handshakes internal
use full-empty interfaces
PhD Dissertation Defense, Swetha Mettala Gilla
slide 41 of 82

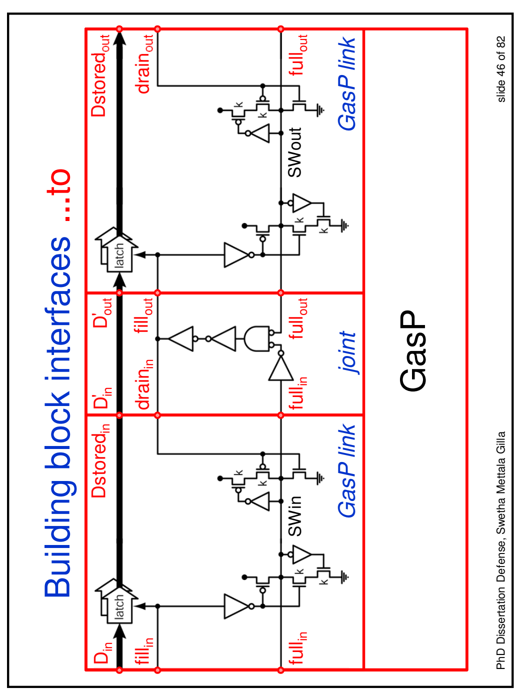

GasP revisited
PhD Dissertation Defense, Swetha Mettala Gilla
slide 42 of 82

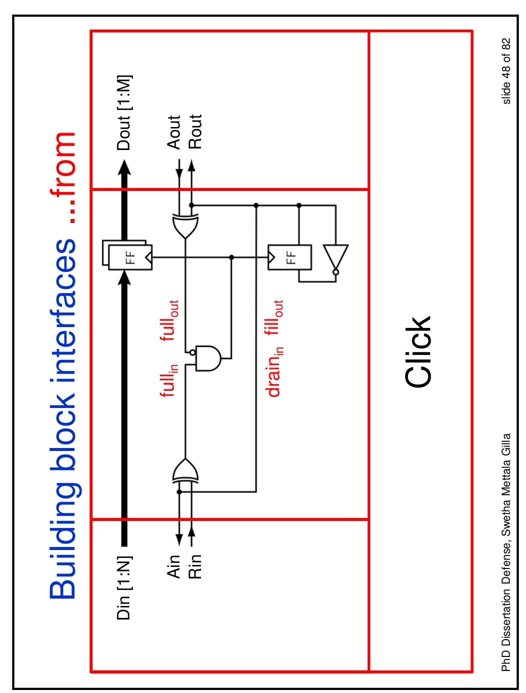

Building block interfaces ...from
Din [1:N]
Dout [1:M]
latch
fill_out
drain_in
full_in
full_out
SWin
SWout
GasP
PhD Dissertation Defense, Swetha Mettala Gilla
slide 43 of 82

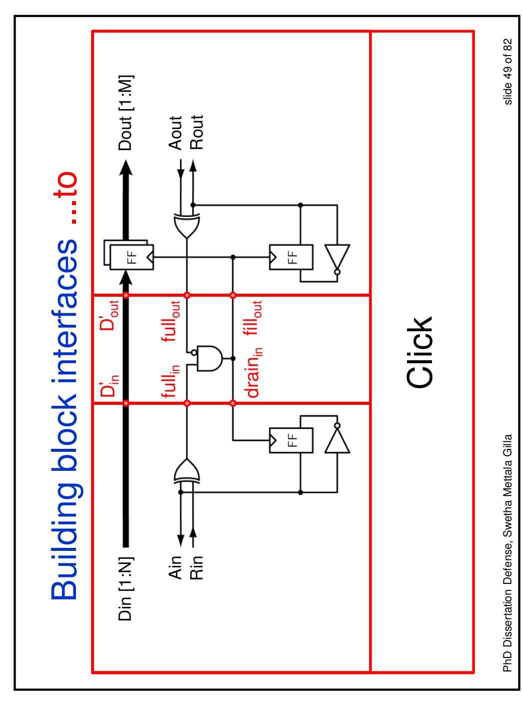

Building block interfaces ...to
Din [1:N]
Dstored$_{out}$
drain$_{out}$
full$_{out}$
SWout
latch
D'$_{out}$
fill$_{out}$
D'$_{in}$
drain$_{in}$
full$_{in}$
SWin
GasP
PhD Dissertation Defense, Swetha Mettala Gilla
slide 45 of 82

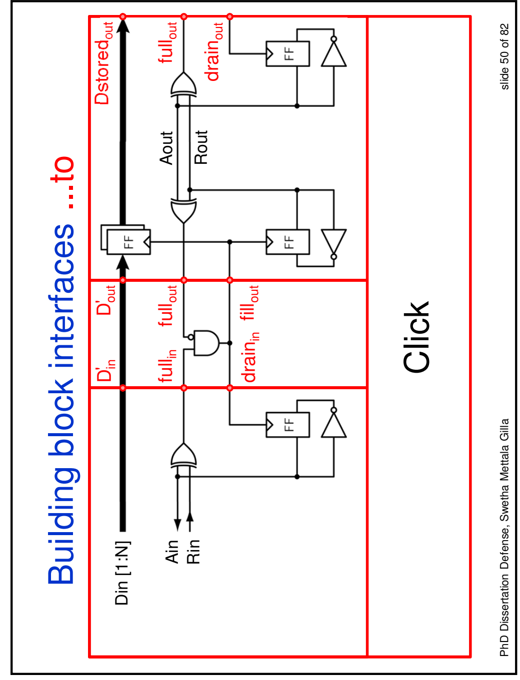

Click revisited
PhD Dissertation Defense, Swetha Mettala Gilla
slide 47 of 82

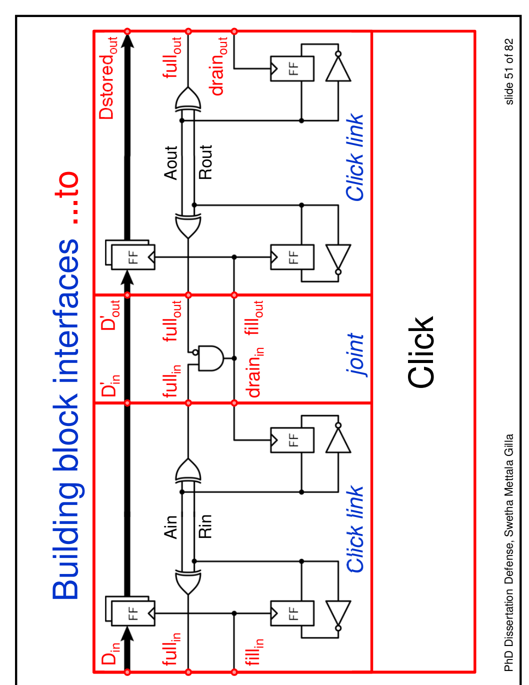

Building block interfaces ...to
Din [1:N]
Dout [1:M]
SWin
SWout
latch
D'$_{in}$
drain$_{in}$
full$_{in}$
D'$_{out}$
fill$_{out}$
full$_{out}$
GasP
PhD Dissertation Defense, Swetha Mettala Gilla
slide 44 of 82

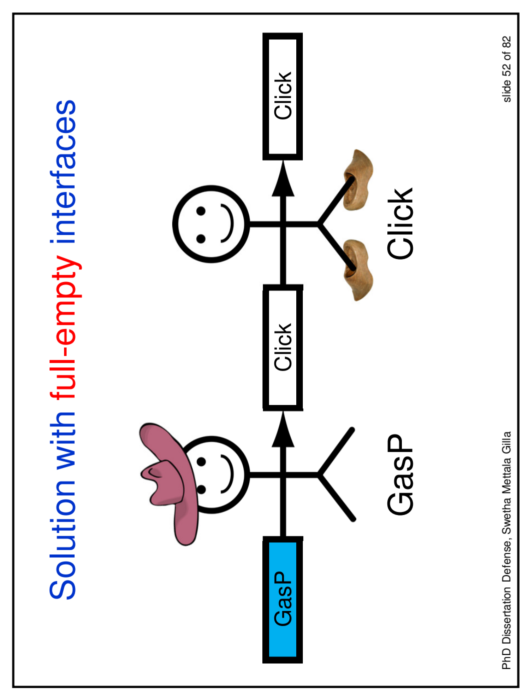

Building block interfaces ...to
D$_{in}$
fill$_{in}$
full$_{in}$
latch
Dstored$_{in}$
SWin
GasP link
D'$_{in}$
drain$_{in}$
full$_{in}$
D'$_{out}$
fill$_{out}$
full$_{out}$
joint
Dstored$_{out}$
drain$_{out}$
full$_{out}$
SWout
GasP link
GasP
PhD Dissertation Defense, Swetha Mettala Gilla
slide 46 of 82

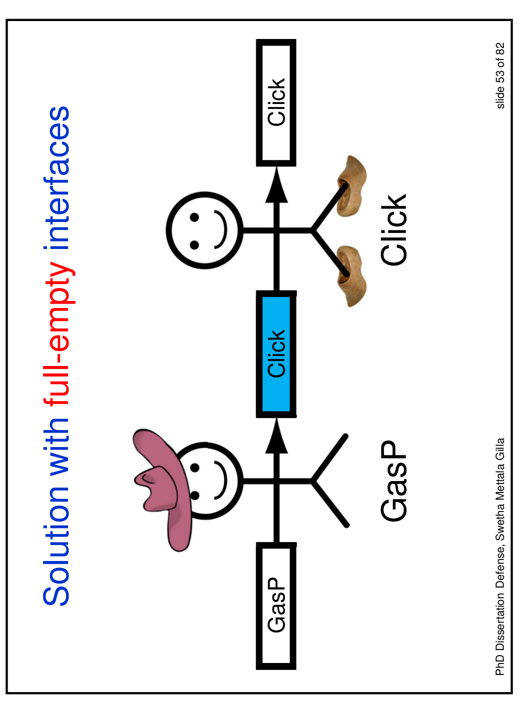

Building block interfaces ...from
Din [1:N]
Ain
Rin
Dout [1:M]
Aout
Rout
FF
FF
full_in
full_out
drain_in
fill_out
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 48 of 82

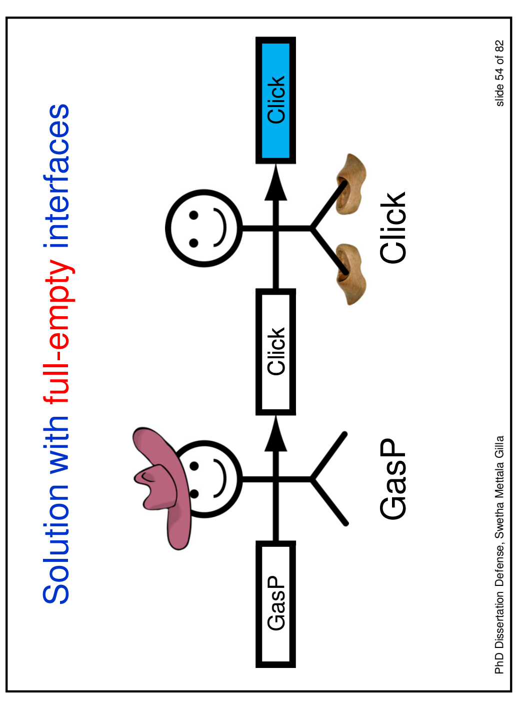

Building block interfaces ...to
Din [1:N]
Ain
Rin
Dout [1:M]
Aout
Rout
D'_in
D'_out
full_in
full_out
drain_in
fill_out
FF
FF
FF
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 49 of 82


Building block interfaces ...to
Din [1:N]
Ain
Rin
D'_in
D'_out
full_in
full_out
drain_in
fill_out
Dstored_out
full_out
drain_out
Aout
Rout
FF
FF
FF
FF
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 50 of 82

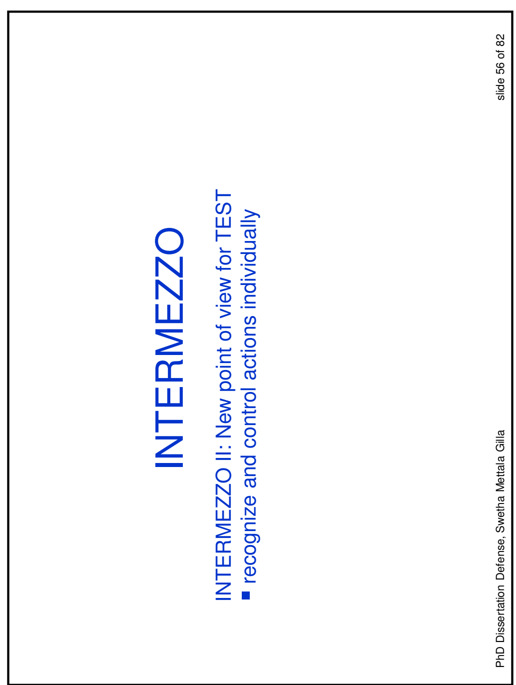

Building block interfaces ...to
D_in
full_in
fill_in
Ain
Rin
Click link
D'_in
D'_out
full_in
full_out
drain_in
fill_out
joint
Aout
Rout
Click link
Dstored_out
full_out
drain_out
FF
FF
FF
FF
FF
FF
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 51 of 82

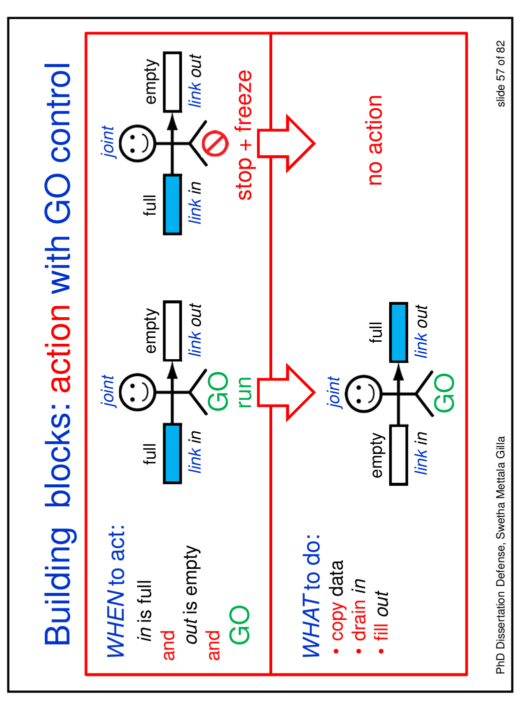

Solution with full-empty interfaces
GasP
Click
Click
GasP
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 52 of 82

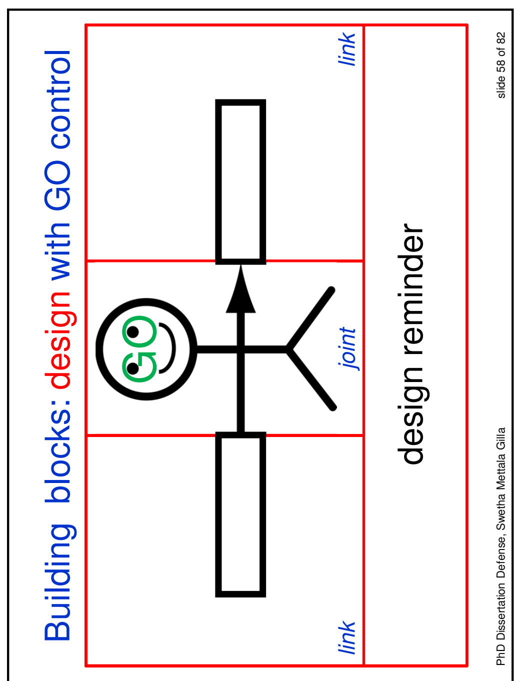

Solution with full-empty interfaces
GasP
Click
Click
GasP
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 53 of 82

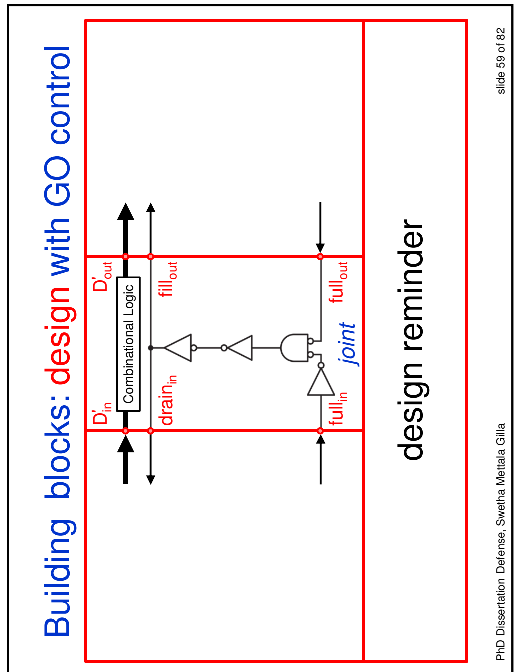

Solution with full-empty interfaces
GasP
Click
Click
GasP
Click
PhD Dissertation Defense, Swetha Mettala Gilla
slide 54 of 82

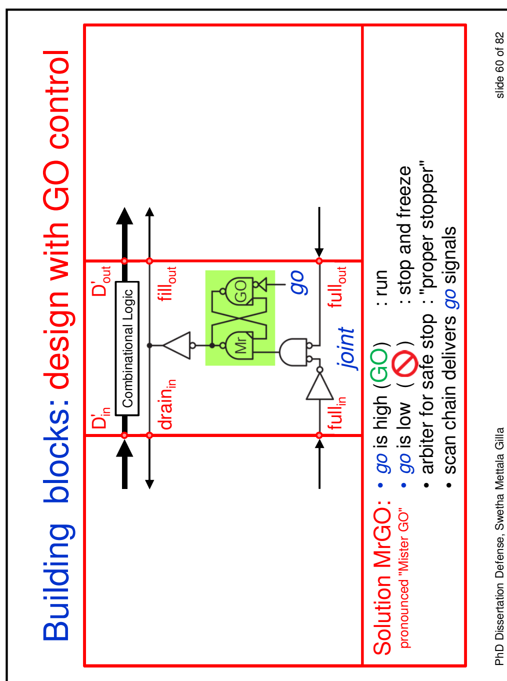
Solution with full-empty interfaces
Provides translation-free communication.
Simplifies collaboration and design re-use.
Please! fill the link
Gotcha! link is full
PhD Dissertation Defense, Swetha Mettala Gilla
slide 55 of 82

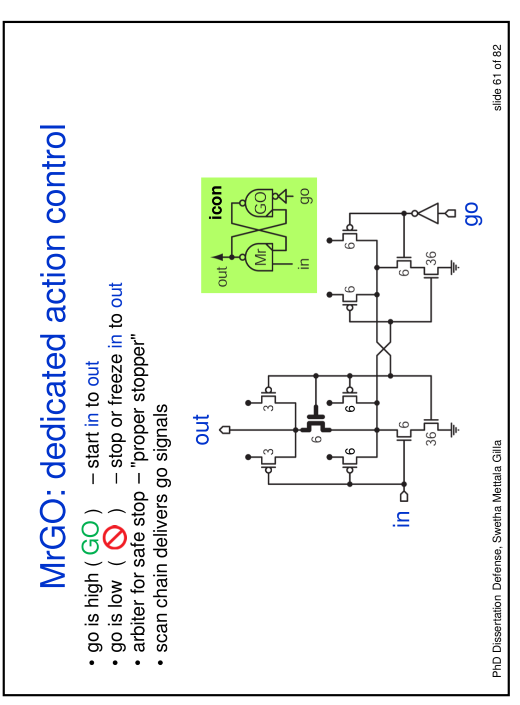

INTERMEZZO
INTERMEZZO II: New point of view for TEST
recognize and control actions individually
PhD Dissertation Defense, Swetha Mettala Gilla
slide 56 of 82

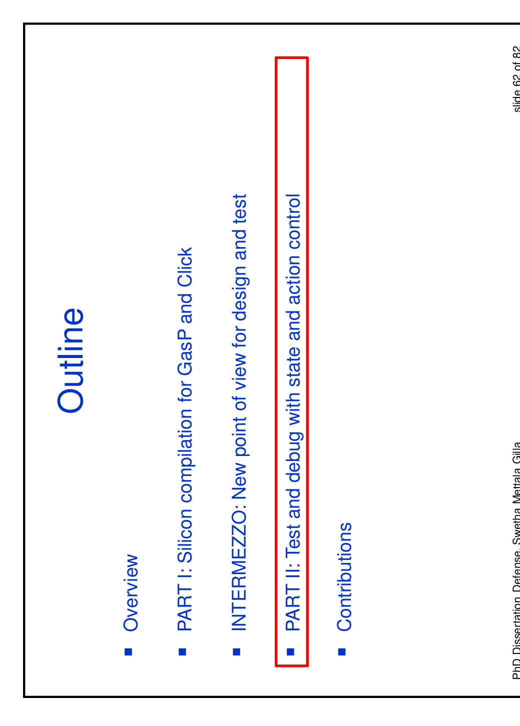

Building blocks: action with GO control
WHEN to act:
in is full
and
out is empty
and
GO
WHAT to do:
• copy data
• drain in
• fill out
joint
full
empty
link in
link out
GO
run
stop + freeze
no action
PhD Dissertation Defense, Swetha Mettala Gilla
slide 57 of 82

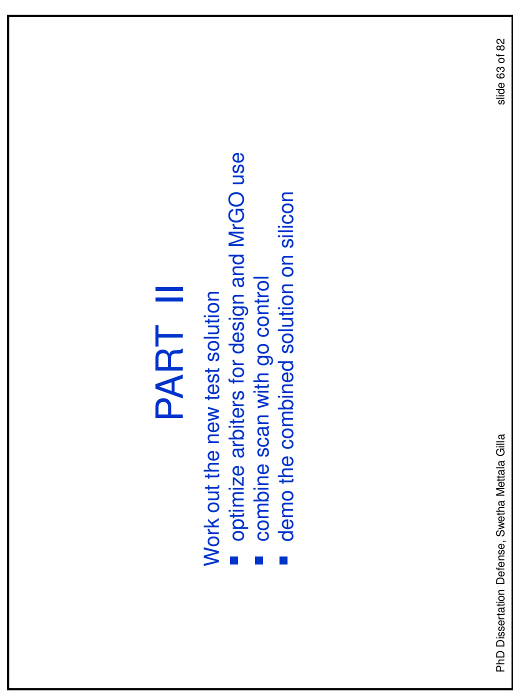

Building blocks: design with GO control
link
joint
link
GO
design reminder
PhD Dissertation Defense, Swetha Mettala Gilla
slide 58 of 82

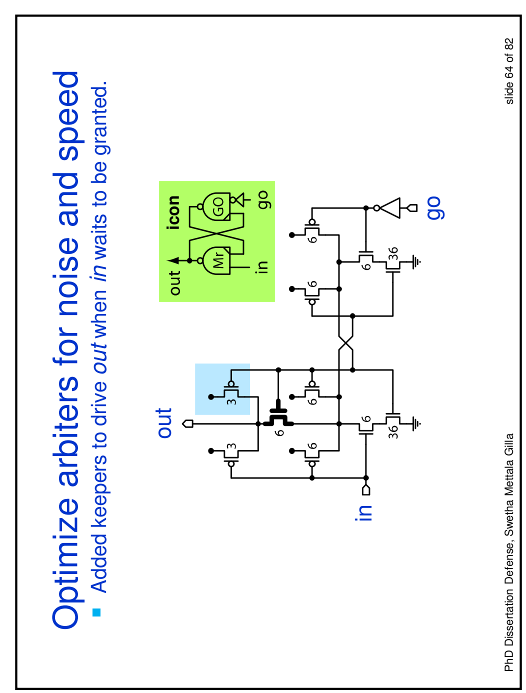

Building blocks: design with GO control
Combinational Logic
joint
design reminder
PhD Dissertation Defense, Swetha Mettala Gilla
slide 59 of 82

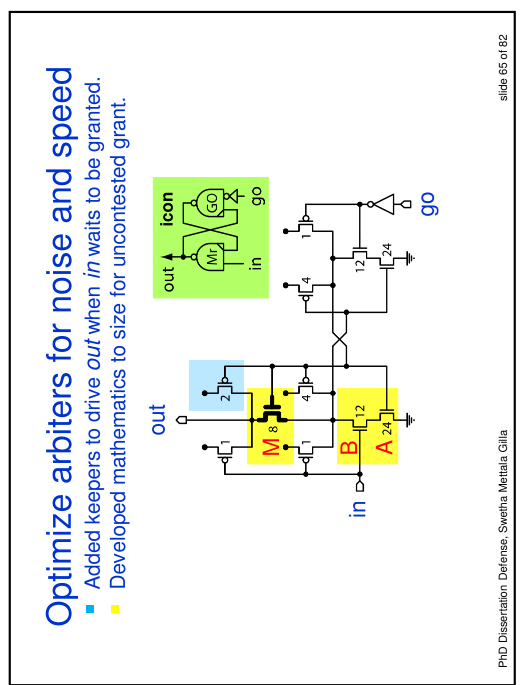

Building blocks: design with GO control
Combinational Logic
drain_in
fill_out
full_in
full_out
joint
go
Mr
GO
Solution MrGO:
pronounced "Mister GO"
• go is high (GO) : run
• go is low ( ) : stop and freeze
• arbiter for safe stop : "proper stopper"
• scan chain delivers go signals
PhD Dissertation Defense, Swetha Mettala Gilla
slide 60 of 82

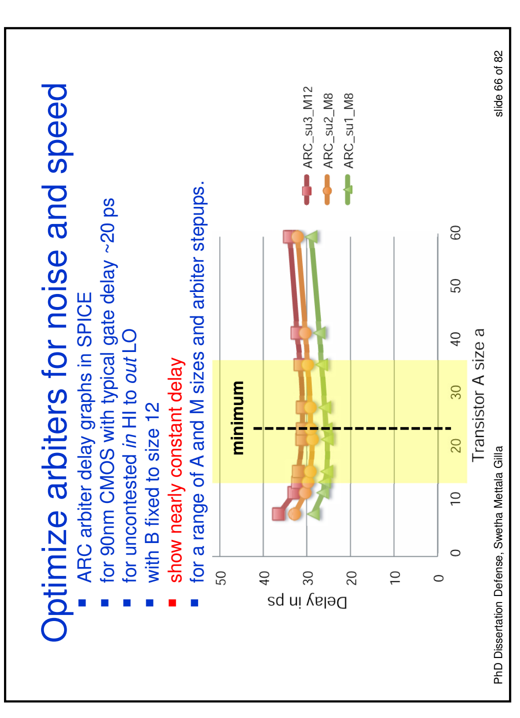

MrGO: dedicated action control
• go is high (GO) – start in to out
• go is low ( ) – stop or freeze in to out
• arbiter for safe stop – "proper stopper"
• scan chain delivers go signals
out
in
go
icon
Mr
GO
PhD Dissertation Defense, Swetha Mettala Gilla
slide 61 of 82

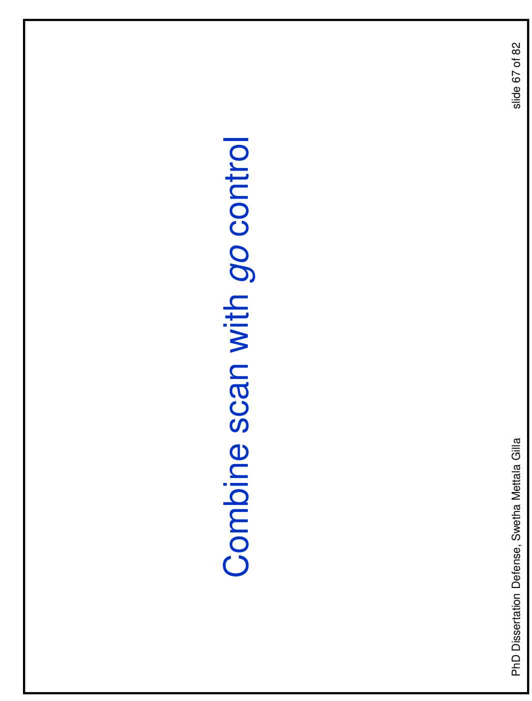

Outline
Overview
PART I: Silicon compilation for GasP and Click
INTERMEZZO: New point of view for design and test
PART II: Test and debug with state and action control
Contributions
PhD Dissertation Defense, Swetha Mettala Gilla
slide 62 of 82

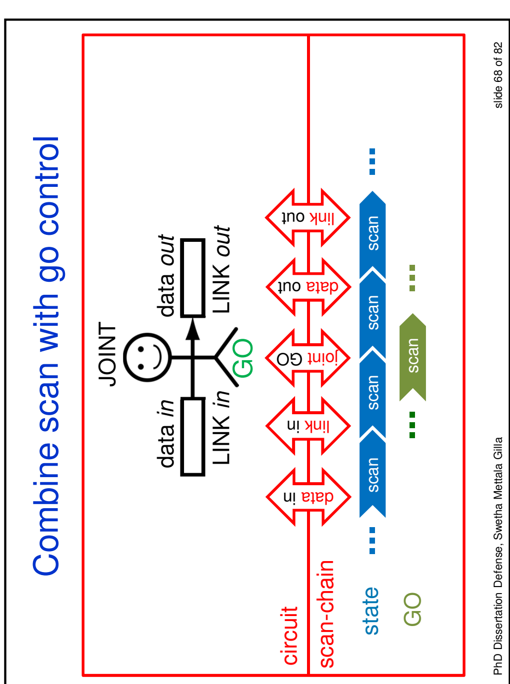

PART II
Work out the new test solution
optimize arbiters for design and MrGO use
combine scan with go control
demo the combined solution on silicon
PhD Dissertation Defense, Swetha Mettala Gilla
slide 63 of 82

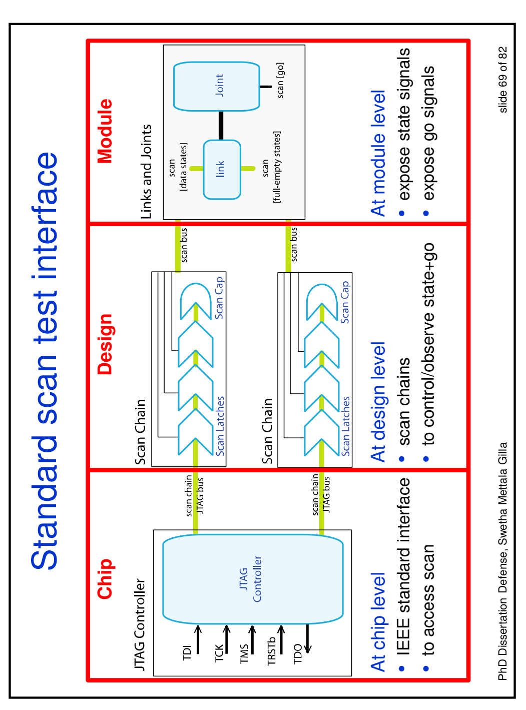

Optimize arbiters for noise and speed
Added keepers to drive *out* when *in* waits to be granted.
out
in
go
icon
out
in
go
Mr
GO
PhD Dissertation Defense, Swetha Mettala Gilla
slide 64 of 82


Optimize arbiters for noise and speed
Added keepers to drive *out* when *in* waits to be granted.
Developed mathematics to size for uncontested grant.
out
in
go
M
B
A
icon
out
in
go
Mr
GO
PhD Dissertation Defense, Swetha Mettala Gilla
slide 65 of 82

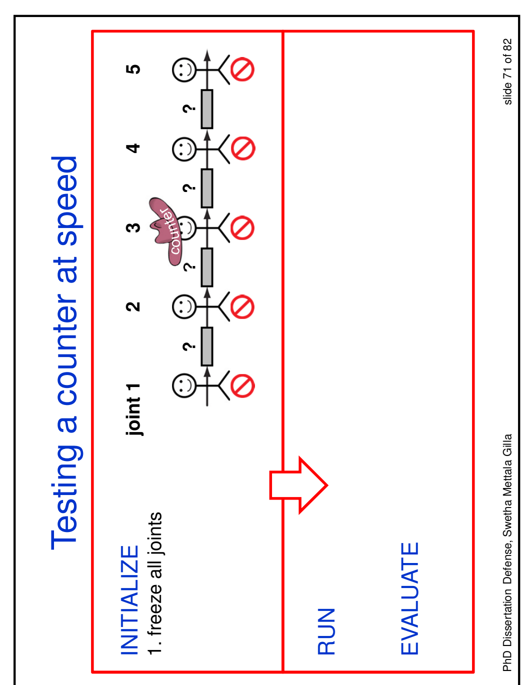

Optimize arbiters for noise and speed
ARC arbiter delay graphs in SPICE
for 90nm CMOS with typical gate delay ~20 ps
for uncontested *in* HI to *out* LO
with B fixed to size 12
show nearly constant delay
for a range of A and M sizes and arbiter stepups.
Delay in ps
Transistor A size a
minimum
ARC_su3_M12
ARC_su2_M8
ARC_su1_M8
PhD Dissertation Defense, Swetha Mettala Gilla
slide 66 of 82

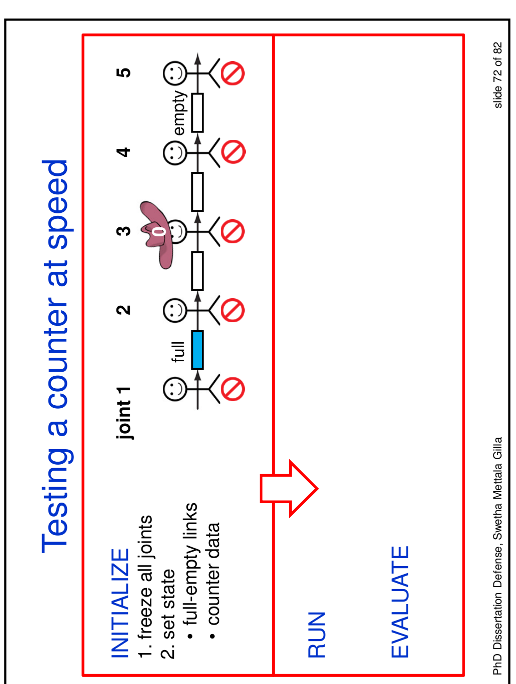

Combine scan with *go* control
PhD Dissertation Defense, Swetha Mettala Gilla
slide 67 of 82

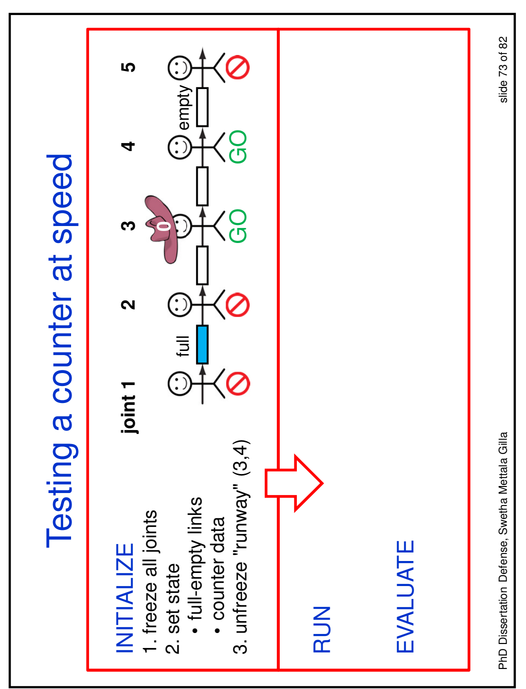

Combine scan with go control
JOINT
data in
LINK in
data out
LINK out
GO
circuit
scan-chain
state
GO
data in
link in
joint GO
data out
link out
scan
scan
scan
scan
scan
PhD Dissertation Defense, Swetha Mettala Gilla
slide 68 of 82

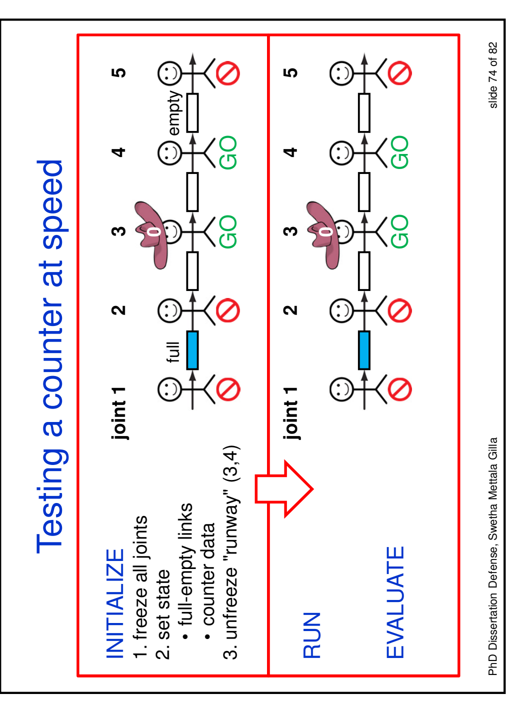

Standard scan test interface
Chip
Design
Module
JTAG Controller
TDI
TCK
TMS
TRSTb
TDO
JTAG Controller
scan chain JTAG bus
Scan Chain
Scan Latches
Scan Cap
scan bus
Links and Joints
link
Joint
scan [go]
At chip level
• IEEE standard interface
• to access scan
At design level
• scan chains
• to control/observe state+go
At module level
• expose state signals
• expose go signals
PhD Dissertation Defense, Swetha Mettala Gilla
slide 69 of 82

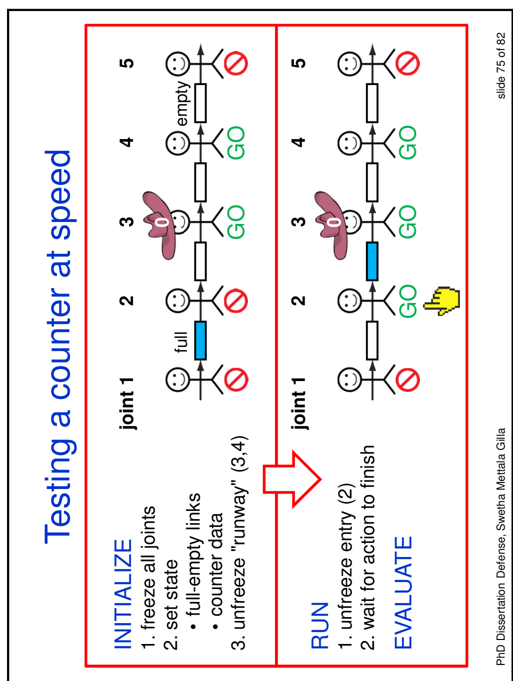
Demo the combined solution on silicon
PhD Dissertation Defense, Swetha Mettala Gilla
slide 70 of 82

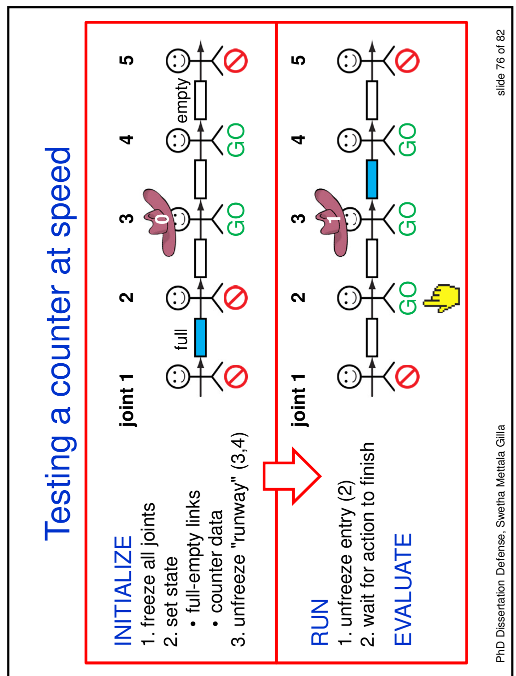

Testing a counter at speed
INITIALIZE
1. freeze all joints
RUN
EVALUATE
joint 1
2
3
4
5
counter
PhD Dissertation Defense, Swetha Mettala Gilla
slide 71 of 82

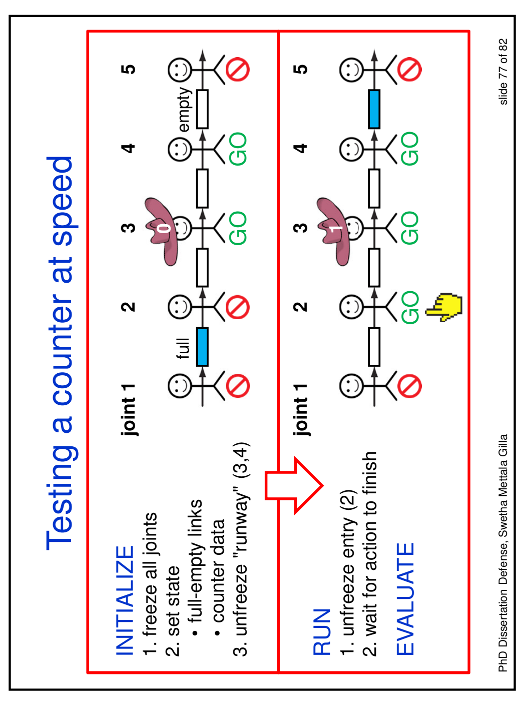

Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
RUN
EVALUATE
joint 1
2
3
4
5
full
empty
0
PhD Dissertation Defense, Swetha Mettala Gilla
slide 72 of 82

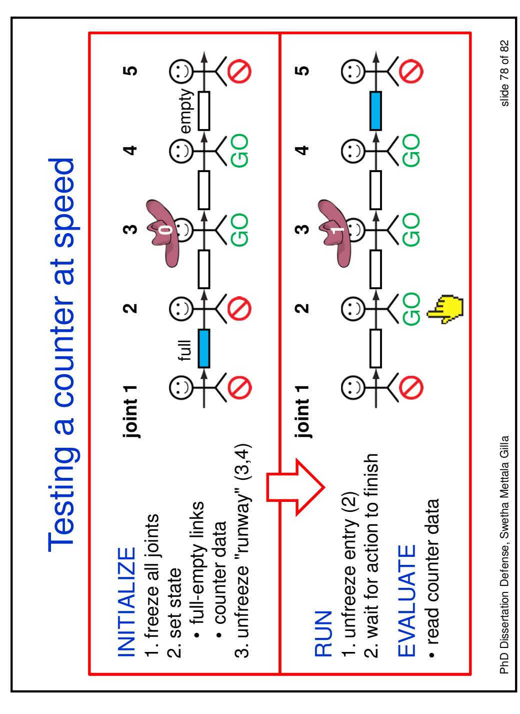

Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
RUN
EVALUATE
joint 1
2
3
4
5
full
empty
0
GO
GO
PhD Dissertation Defense, Swetha Mettala Gilla
slide 73 of 82


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
RUN
EVALUATE
joint 1
2
3
4
5
full
empty
0
GO
GO
PhD Dissertation Defense, Swetha Mettala Gilla
slide 74 of 82


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
full
empty
0
GO
GO
GO
PhD Dissertation Defense, Swetha Mettala Gilla
slide 75 of 82


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
full
empty
GO
0
1
PhD Dissertation Defense, Swetha Mettala Gilla
slide 76 of 82


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
joint 1
2
3
4
5
full
empty
GO
0
1
PhD Dissertation Defense, Swetha Mettala Gilla
slide 77 of 82


Testing a counter at speed
INITIALIZE
1. freeze all joints
2. set state
• full-empty links
• counter data
3. unfreeze "runway" (3,4)
RUN
1. unfreeze entry (2)
2. wait for action to finish
EVALUATE
• read counter data
joint 1
2
3
4
5
full
empty
GO
0
1
PhD Dissertation Defense, Swetha Mettala Gilla
slide 78 of 82

GET REAL
MrGO approved
Two working silicon experiments: Weaver and Anvil
use building blocks with full-empty interfaces
and MrGO + JTAG-scan for test, debug, characterization.
marly
swetha
hoon
nav
chris
ivan
PhD Dissertation Defense, Swetha Mettala Gilla
slide 79 of 82

My contributions (reminder)

System design
- My research influenced ARC's new design and test point of view.
- I created Telescope GasP – a GasP extension still relevant today.
- I extended ARCwelder to support (T)GasP.

Arbitrated circuits
- With MrGO, arbiters are everywhere now.
- I improved the noise tolerance for arbiter inputs and outputs.
- I created the mathematical foundation to size arbiters for speed.

Test, debug, and performance characterization
- I implemented MrGO and scan.
- I built and demo-ed the combined MrGO-scan solution on silicon.
- I proposed an initialization solution which works at power-up.

PhD Dissertation Defense, Swetha Mettala Gilla slide 80 of 82

PhD Dissertation Defense, Swetha Mettala Gilla slide 81 of 82

Asynchronous Research Center (ARC)

Asynchronous Research Center, Portland State University, 1900 SW 4th Ave, FAB 105, Portland, OR 97201, USA

PhD Dissertation Defense, Swetha Mettala Gilla slide 82 of 82