

Design and Test of Asynchronous Systems
Using the Link and Joint Model

by

Ebelechukwu Esimai

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

Doctor of Philosophy
in
Computer Science

Dissertation Committee:
Marly Roncken, Chair
Andrew Tolmach
Mark P. Jones
Xiaoyu Song
Gary Delp

Portland State University
2024

© 2024 Ebelechukwu Esimai

Abstract

Asynchronous circuits offer numerous advantages, including low energy consumption and good composability and scalability. However, they remain meagerly adopted in the mainstream semiconductor industry. One reason is the limited number of design tools available to help designers navigate design complexity, particularly the myriad of asynchronous implementation styles.

This dissertation focuses on managing the myriad of asynchronous implementation styles by utilizing a circuit-neutral model, called *Links* and *Joints*, and embedding this Link-Joint approach into a design flow. Although years of past work have already laid the groundwork, the work in this dissertation identifies and addresses key missing pieces.

First, the dissertation presents a design and test methodology centered around Links and Joints that exploits the similarities between multiple circuit implementation styles. This methodology offers interface uniformity and generality for various asynchronous circuit families and protocols, as well as flexibility in implementation choices and circuit initialization.

Second, this dissertation shows the Link-Joint methodology embedded in a design flow. The resulting flow, called *Ọnà* (/or-NUHR/, Yoruba for “way”), includes compilation and refinement steps for transforming high-level parallel programs with message passing via circuit-neutral Link-Joint networks into asynchronous circuits, postponing choices in protocol and circuit family as late as possible. *Ọnà* also carries

along test and debug, using a uniform test approach that fosters test reuse from one abstraction level to another.

Qnà makes it easy to insert asynchrony appropriate for each design part. The dissertation demonstrates this ease by providing methodology and design flow support for various protocols such as 2- and 4-phase protocols, level- and pulse- and transition-signaling logic, bundled data, and circuit families such as Click, GasP, Set-Reset, Mousetrap, Micropipelines, and the Single Flux Quantum (*SFQ) superconductor family. The dissertation also demonstrates that mixing and matching different circuit implementation styles in *Qnà* is flexible and straightforward.

Acknowledgments

This dissertation would not have been possible without the help of many people, and they have my sincere gratitude. First and foremost, I sincerely appreciate my advisor, Marly Roncken, for her guidance, invaluable insights, and unwavering support throughout this journey. Her expertise, encouragement, and mentorship have been instrumental in shaping my research endeavors and my person.

I give special thanks to my dissertation committee members, Professor Andrew Tolmach, Professor Mark Jones, Professor Xiaoyu Song, and Gary Delp, for their patience, feedback, and guidance in shaping this dissertation. Also, I appreciate Rajit Manohar for providing open access to the Asynchronous Circuit Toolkit (ACT) and for his invaluable collaboration and feedback.

I am immensely grateful to Ivan Sutherland for his encouragement, insight into asynchronous circuit design, and feedback. I give many thanks to ARC members, both past and present, for laying the foundation for this work. I would also like to acknowledge Gary Delp and the Mayo Team for their insightful feedback, which has greatly enriched the quality and scope of this research. I extend my heartfelt thanks to Warren Hunt for his pivotal role in connecting me with my advisor and guiding me toward my research area.

Finally, I have immeasurable gratitude for my family (my sisters Nwando and Ileri, my mum Peju, my partner Obinna, my uncles and aunties) and my friends (Toso, Lakshmi, Kata, and PSU Jollof crew). Your support and care have helped me through this journey. Thank you!

Table of Contents

| | |
|--|------|
| Abstract | i |
| Acknowledgments | iii |
| List of Figures | viii |
| List of Abbreviations | xiv |
| Chapter 1: Introduction | 1 |
| 1.1 Research Summary | 2 |
| 1.1.1 Proposed Design Flow: <i>Qnà</i> | 3 |
| 1.1.2 Contributions | 5 |
| 1.1.3 Publications Based on this Work | 6 |
| 1.2 Dissertation Organization | 7 |
| Chapter 2: Background | 9 |
| 2.1 Communication Protocols | 10 |
| 2.1.1 Handshakes | 10 |
| 2.1.2 Signaling Logic | 13 |
| 2.1.3 Data Encodings | 14 |
| 2.2 Asynchronous Circuit Families | 15 |
| 2.3 Design Approaches | 17 |
| 2.3.1 Compilation-Based Design Flows | 18 |
| 2.4 Test and Debug | 21 |
| Chapter 3: Links and Joints | 23 |
| 3.1 The Link-Joint Model | 23 |
| 3.2 Shared Variable Semantics | 26 |
| 3.2.1 Semantics in Action | 30 |
| 3.3 Flexible Initialization | 33 |

| | |
|---|-----|
| 3.4 Executable model in Verilog | 39 |
| 3.5 Chapter Contributions | 43 |
| Chapter 4: Compilation | 46 |
| 4.1 Source Programs | 47 |
| 4.1.1 The Communicating Hardware Processes (CHP) Sublanguage | 47 |
| 4.1.2 The Dataflow Sublanguage | 49 |
| 4.2 Link-Joint Library Elements | 50 |
| 4.2.1 Computation Class - Joints <i>VAR, E, TRF</i> | 52 |
| 4.2.2 Shared Resources Class - Joints <i>RMUX, WMUX, MUX, FSM</i> | 54 |
| 4.2.3 Flow Control Class - Joints <i>SEQ, PAR, REP, SEL</i> | 56 |
| 4.2.4 Communication Class - Joints <i>CHAN, E_{waitcycle}, WAIT_{cycle}</i> . . | 59 |
| 4.2.5 Dataflow Class - Joints <i>COPY, SRC, SNK, FORK, RROT</i> . . | 62 |
| 4.3 Compilation to Links and Joints | 63 |
| 4.3.1 ACT Compiler Modifications | 68 |
| 4.3.2 Compiler Optimization | 71 |
| 4.4 Chapter Contributions | 72 |
| Chapter 5: Refinement | 74 |
| 5.1 Asynchronous Design Styles | 75 |
| 5.2 Refinement Examples | 76 |
| 5.2.1 Data Storage Refinement: To Store or Not to Store | 76 |
| 5.2.2 Atomicity Refinement: Fuse or Split Joint Atomicity | 81 |
| 5.2.3 Nondeterministic Selection Implementation | 85 |
| 5.3 Abstract Gate-level Implementations | 88 |
| 5.3.1 2-phase level-signaling bundled data Link circuits | 88 |
| 5.3.2 Circuit implementations of Joint <i>COPY</i> | 92 |
| 5.3.3 4-phase level-signaling bundled data Link circuits | 94 |
| 5.4 Chapter Contributions | 96 |
| Chapter 6: Test, Debug, and Initialization | 98 |
| 6.1 Existing State-Action Test and Debug Approach | 99 |
| 6.1.1 Connecting the Existing Test Solution to the Program Level . | 102 |
| 6.2 Functional Testing – Asynchronous Ring FIFO example | 104 |
| 6.2.1 Program Level | 104 |
| 6.2.2 Link-Joint Level | 106 |
| 6.2.3 Ring FIFO Throughput Simulation | 107 |
| 6.2.4 Circuit Level | 111 |

| | | |
|--|--|-----|
| 6.3 | Structural Testing | 113 |
| 6.3.1 | Assignment | 116 |
| 6.3.2 | Loops and Selection | 119 |
| 6.3.3 | Communication | 123 |
| 6.4 | Connecting Our Uniform Test Approach to Hardware Test Methods . . | 125 |
| 6.5 | Chapter Contributions | 125 |
| Chapter 7: From Link-Joint Circuits to ACT | | 127 |
| 7.1 | <i>Qnà</i> — A Shallow Link-Joint Embedding in ACT | 127 |
| 7.2 | Back into ACT with Gates and Timing | 129 |
| 7.2.1 | Production Rule Set (PRS) | 129 |
| 7.2.2 | Timing Constraints | 130 |
| 7.3 | Gate-level Link-Joint example in ACT | 132 |
| 7.3.1 | Link Implementation for <i>Ring2</i> | 134 |
| 7.3.2 | Joint Implementation for <i>Ring2</i> | 137 |
| 7.3.3 | Link-Joint network Implementation for <i>Ring2</i> | 139 |
| 7.3.4 | Final Notes on Timing Constraints | 142 |
| 7.4 | Chapter Contributions | 142 |
| Chapter 8: Mixing and Matching Circuit Implementation Styles | | 145 |
| 8.1 | Benefits of Various Asynchronous Circuit Implementation Styles | 145 |
| 8.2 | Example: A Mixed Ring FIFO In Action | 148 |
| 8.3 | Example: Mixing Asynchronous Circuit Families | 149 |
| 8.4 | Example: Mixing Protocols | 151 |
| 8.5 | Example: Mixing Signaling Logic | 152 |
| 8.6 | Matching Implementations to Links and Joints | 154 |
| 8.7 | Chapter Contributions | 158 |
| Chapter 9: Conclusion and Future Work | | 159 |
| 9.1 | Future Research Directions | 160 |
| Bibliography | | 162 |
| Appendices | | 169 |
| Appendix A: Test Setup for <i>Ring2</i> in Chapter 7 | | 170 |

| | |
|---|-----|
| Appendix B: Waveforms for Mixed Implementation <i>Ring6</i> | 176 |
|---|-----|

List of Figures

| | | |
|-----|--|----|
| 1.1 | Our design flow, called <i>Ọnà</i> (/or-NUHR/, Yoruba for “way”). | 3 |
| 2.1 | A typical 2-phase handshake protocol with <i>request</i> and <i>acknowledge</i> signals. | 11 |
| 2.2 | A typical 4-phase handshake protocol with request and acknowledge signals with bundled data and a typical 4-phase handshake protocol with dual-rail data and acknowledge signal. | 12 |
| 2.3 | 2-phase handshake protocol depicted with <i>request</i> and <i>acknowledge</i> , <i>statewire</i> , pulse logic <i>request</i> and <i>acknowledge</i> | 14 |
| 2.4 | Handshake circuits: Intermediary in the Tangram design flow. | 20 |
| 3.1 | A simple Link-Joint network that adds two numbers as a data-flow computation with channels, $A + B \rightarrow X$ | 24 |
| 3.2 | A simple Link-Joint network that adds two numbers as a control-flow computation with variables, $x := a + b$ | 25 |
| 3.3 | Protocol and model of a Link (a) and a Joint (b). | 27 |
| 3.4 | Example Link-Joint network with two Links, $L1$ and $L2$, and one Joint <i>COPY</i> (a), with a guarded command specification for the Joint (b), and an example interpretation of the Joint’s terms in relation to its connected Links (c). | 30 |
| 3.5 | Ping-Pong Link-Joint network with a Link and two <i>SKIP</i> Joints with the guarded command specification of Joint <i>SKIP</i> | 30 |
| 3.6 | Initializing the Ping-Pong Link-Joint network. | 31 |
| 3.7 | The two states that the Ping-Pong goes back and forth on. | 32 |
| 3.8 | Final state of the Ping-Pong run, after we make the <i>go</i> signal in Joint $J1$ FALSE. | 33 |
| 3.9 | A simple FIFO with unspecified active-passive protocol. | 34 |

| | | |
|------|--|----|
| 3.10 | The simple FIFO in figure 3.9 with four different initial active-passive protocol setting configuration. | 35 |
| 3.11 | Snippet of execution from the initial active-passive configuration in Figure 3.10(a). | 36 |
| 3.12 | Snippet of execution from the initial active-passive configuration in Figure 3.10(b). | 36 |
| 3.13 | Snippet of execution from the initial active-passive configuration in Figure 3.10(c). | 38 |
| 3.14 | Snippet of execution from the initial active-passive configuration in Figure 3.10(d). | 38 |
| 3.15 | Verilog behavioral module for a Link. | 40 |
| 3.16 | Verilog behavioral module for Joint <i>COPY</i> | 42 |
| 3.17 | A Verilog testbench for a simple FIFO (Figure 3.9). | 44 |
| 3.18 | Waveforms for simulating Verilog testbench in Figure 3.17. | 45 |
| 4.1 | Compilation as a focus in our design flow, <i>Onà</i> | 47 |
| 4.2 | Joint VAR library element. | 52 |
| 4.3 | Joint E library element. | 53 |
| 4.4 | Joint TRF library element. | 54 |
| 4.5 | Joint RMUX library element. | 55 |
| 4.6 | Joint WMUX library element. | 56 |
| 4.7 | Joint MUX library element. | 56 |
| 4.8 | Joint SEQ library element. | 57 |
| 4.9 | Joint PAR library element. | 57 |
| 4.10 | Joint REP library element. | 58 |
| 4.11 | Joint SEL library element representing SEL_{det} and SEL_{nondet} | 59 |
| 4.12 | Joint CHAN library element. | 60 |
| 4.13 | Joint $E_{waitcycle}$ library element. | 61 |

| | | |
|------|--|----|
| 4.14 | Joint $WAIT_{cycle}$ library element. | 62 |
| 4.15 | An ACT dataflow program for a two-stage FIFO buffer (top) with its corresponding compiled Link-Joint network (bottom). | 64 |
| 4.16 | An ACT hierarchical CHP program for a two-stage FIFO with its corresponding compiled Link-Joint network. | 65 |
| 4.17 | Our compiler output for the two-stage FIFO example in Figure 4.16. | 67 |
| 5.1 | Refinement as a focus in the <i>Qnà</i> design flow. | 75 |
| 5.2 | Link-Joint network representing two parallel program fragments that exchange data between both sides. | 77 |
| 5.3 | Link-Joint network showing the first refinement solution for example in Figure 5.2. | 78 |
| 5.4 | Link-Joint network showing the second refinement solution, for example, in Figure 5.2. | 79 |
| 5.5 | Joint F library element. | 82 |
| 5.6 | Refined Joint SEL library element representing SEL_{nondet} with an arbiter guard selection implementation. | 85 |
| 5.7 | Refined Joint SEL library element representing SEL_{nondet} with a round-robin guard selection implementation. | 86 |
| 5.8 | Refined Joint SEL library element representing SEL_{nondet} with a static or fixed priority guard selection implementation. | 87 |
| 5.9 | A Link with a <i>2-phase level-signaling bundled-data</i> protocol in the Click asynchronous family | 89 |
| 5.10 | A Link with a <i>2-phase level-signaling bundled-data</i> protocol and data storage in the Set-Reset asynchronous family. | 90 |
| 5.11 | A Link with a <i>2-phase level-signaling bundled-data</i> protocol and data storage in the GasP asynchronous family. | 91 |
| 5.12 | Waveforms showing the pattern of interface signals, internal signals, and data validity for a bidirectional Link with a <i>2-phase level-signaling bundled-data</i> protocol and data storage. | 92 |
| 5.13 | Joint <i>COPY</i> gate implementation. | 93 |

| | | |
|------|--|-----|
| 5.14 | A Link with <i>4-phase level-signaling bundled-data</i> protocol and data storage in Click (a), GasP (b), and Set-Reset (c) asynchronous families. | 94 |
| 5.15 | Data validity for a unidirectional <i>4-phase</i> Link with communication data split into <i>early</i> and <i>late</i> . | 96 |
| 6.1 | Our uniform test approach harmonizes program interactive code debug, Link-Joint action-state control, and circuit and chip level Design-for-Test (DfT) techniques such as scan test. | 99 |
| 6.2 | A Link-Joint network for a two-stage FIFO with one Joint, <i>COPY</i> , and two Links. | 100 |
| 6.3 | A circuit for a two-stage FIFO using a <i>2-phase bundled-data</i> protocol in the Set-Reset asynchronous circuit family. | 101 |
| 6.4 | The one-to-one relation between Link variables and Joint <i>go</i> -control and circuit-level scan chains. | 101 |
| 6.5 | A CHP program for a one-stage FIFO buffer from Figure 4.16. | 102 |
| 6.6 | ACT program for an asynchronous ring FIFO with a storage capacity of up to four 4-bit data items. | 105 |
| 6.7 | The Link-Joint network for an asynchronous ring FIFO with a storage capacity of up to four data items and its external test access. | 106 |
| 6.8 | Sample ACTSIM script to initialize and simulate the ring FIFO at the program level, using the CHP program in the right panel of Figure 6.6. | 108 |
| 6.9 | Sample Verilog testbench to initialize and simulate the ring FIFO at Link-Joint level, using the Link-Joint network in Figure 6.7. | 109 |
| 6.10 | Canopy graph showing the simulated throughput of the ring FIFO with varying numbers of data items at different abstraction levels. | 110 |
| 6.11 | A <i>2-phase bundled-data pulse-logic</i> RSFQ (superconducting family) implementation of a Link and Joint <i>COPY</i> . | 111 |
| 6.12 | RSFQ ring FIFO implementation with reduced scan access (top) and an associated canopy graph showing the simulated throughput (bottom). | 113 |
| 6.13 | A CHP program for the Greatest Common Divisor (GCD) algorithm, augmented with labels and <i>log</i> statements for use in debug and test generation. | 114 |

| | | |
|------|--|-----|
| 6.14 | A Link-Joint network for the Greatest Common Divisor (GCD) algorithm, compiled from the CHP program in Figure 6.13, using the compilation approach explained in Chapter 4. | 115 |
| 6.15 | Program Assignment test sequence as an ACTSIM script and its output. | 117 |
| 6.16 | Link-Joint Assignment test sequence as a Verilog testbench. | 118 |
| 6.17 | Waveforms from the simulation of the Assignment Verilog testbench. | 119 |
| 6.18 | Program Select-Loop test sequence as an ACTSIM script and its output. | 120 |
| 6.19 | Link-Joint Select-Loop test sequence as a Verilog testbench. | 122 |
| 6.20 | Waveforms from the simulation of the Select-Loop Verilog testbench. | 122 |
| 6.21 | Program test sequence to test output communication $O!x$ in isolation with ACTSIM. | 123 |
| 6.22 | Link-Joint test sequence to test output communication $O!x$ in isolation as a Verilog testbench. | 124 |
| 6.23 | Waveforms from the simulation of the Verilog testbench in Figure 6.22 | 125 |
| 7.1 | Our design and test flow, $Qnà$, is implemented as a shallow embedding in the ACT ecosystem. | 128 |
| 7.2 | PRS and Verilog specifications of a two input AND gate. | 129 |
| 7.3 | PRS and Verilog specifications of a two input C -element gate. | 130 |
| 7.4 | Link-Joint network for a 2-stage ring, $Ring2$, with $RROT$ Joints. | 132 |
| 7.5 | Circuit representation of $Ring2$ design using $Click$ Links and 2 -phase level-signaling bundled-data protocol showing all its signal names. | 133 |
| 7.6 | PRS specifications of gates used in Link and Joint modules in $Ring2$. | 133 |
| 7.7 | PRS specifications of gates used in Link and Joint modules in $Ring2$. | 134 |
| 7.8 | ACT-PRS specification for a 2-phase level-signaling bundled-data Click Link with timing constraints typical for Click circuits. | 135 |
| 7.9 | Visual depiction of timing forks in the $Click$ Link as paths. | 136 |
| 7.10 | ACT specification for a 2-phase level-signaling bundled-data $RROT$ Joint with timing constraints typical for 2-phase level signaling. | 138 |

| | | |
|------|--|-----|
| 7.11 | Visual depiction of timing constraints in the <i>RROT</i> Joint. | 139 |
| 7.12 | ACT specification for a 2-phase level-signaling bundled-data 2-stage ring with Joint <i>RROT</i> for 3-bit data with timing constraints typical for handing over data in the ring. | 140 |
| 7.13 | Simulation waveform <i>Ring2</i> , rotating and circulating one data item, starting with $B_myR[2:0] = 3'b001$ in Link $L[0]$ | 141 |
| 7.14 | Visual depiction of timing paths in the <i>Ring2</i> design governing data handover. | 144 |
| 8.1 | Link-Joint network for a 6-stage mixed ring FIFO, using both <i>2-phase</i> and <i>4-phase</i> protocols as well as <i>Set-Reset</i> and <i>Click</i> circuit families. | 148 |
| 8.2 | Canopy graph of the 6-stage mixed ring FIFO in Figure 8.1. | 149 |
| 8.3 | Mixed circuit implementation with a <i>Set-Reset</i> Link (left), Joint <i>COPY</i> (middle), and a <i>Click</i> Link (right). | 150 |
| 8.4 | Mixed circuit implementation for transition signaling with a <i>Micropipeline</i> Link (left), Joint <i>COPY</i> (middle), and a <i>Mousetrap</i> Link (right). | 151 |
| 8.5 | Bidirectional Links in circuit families (a) <i>Click</i> , (b) <i>GasP</i> , and (c) <i>Set-Reset</i> using <i>2-phase</i> at Link port <i>A</i> , and <i>4-phase</i> at Link port <i>B</i> | 152 |
| 8.6 | Mixed circuit implementation where a <i>2-phase</i> Joint (left) and a <i>4-phase</i> Joint (right) are connected with a <i>4-to-2 phase GasP</i> Link and a <i>2-to-4 phase Click</i> Link. This Link-Joint FIFO also functions correctly when connected as a ring FIFO. | 152 |
| 8.7 | Mixed circuit implementation where a <i>2-phase level-signaling</i> Joint (left) and a <i>transition</i> Joint (right) are connected with a <i>transition to 2-phase level signaling</i> Link, <i>L1</i> , based on <i>Mousetrap</i> and <i>Click</i> , and a <i>2-phase level to transition signaling</i> Link, <i>L2</i> , based on <i>Click</i> | 153 |
| 8.8 | Mixed circuit implementation where a <i>4-phase level-signaling</i> Joint (left) and a <i>transition</i> Joint (right) are connected with a <i>transition to 4-phase level signaling</i> Link, <i>L1</i> , based on <i>Mousetrap</i> and <i>Click</i> , and a <i>4-phase level to transition signaling</i> Link, <i>L2</i> , based on <i>Click</i> | 154 |
| 8.9 | RSFQ dual-rail design. | 155 |
| 8.10 | RSFQ gates used in the dual-rail design in Figure 8.9. | 155 |
| 8.11 | Link-Joint partition for the design in Figure 8.9. | 157 |

List of Abbreviations

ACT Asynchronous Circuit Toolkit

CAD Computer Aided Design

CHP Communicating Hardware Processes

CSP Communicating Sequential Processes

DfT Design for Testability

EDA Electronic Design Automation

FIFO First In First Out

NDRO Nondestructive Readout

Non-RTZ Non-Return-to-Zero

POD Point-of-Divergence

PRS Production Rule Set

QDI Quasi-Delay-Insensitive

RSFQ Rapid Single Flux Quantum

RTZ Return-to-Zero

STG Signal Transition Graph

VLSI Very Large Scale Integration

Chapter 1: Introduction

Asynchronous design has remained a niche in the semiconductor industry for several decades despite advantages such as efficient energy consumption, improved throughput, enhanced robustness, reliability, and increased scalability. However, with more recent technological advances and a wider range of applications such as wearable devices, biomedical implants, and autonomous systems, there is renewed interest in using asynchronous circuits. To support both experienced and new designers in the asynchronous design space, we are revisiting its design and test methodology.

An asynchronous (or self-timed) circuit has no *global clock*. Instead, it uses local protocols between its components to govern communication, synchronization, and sequencing operations [66]. There are many ways to implement the local communication, protocols, and data operations needed in the circuit. When grouped and optimized for a specific goal, different circuit implementation styles form diverse asynchronous circuit families. These families are alike in every intent but different in detail. The differences in initialization, handshake signaling, and static timing make it challenging to work with or combine multiple families at once. Differences among asynchronous circuit families also make it complex to design and test both general and mixed implementations of asynchronous families with the support of electronic design automation (EDA) tools — tools that the synchronous world has the privilege of in abundance but are sparse in the asynchronous world.

In this dissertation, we aim to (1) showcase a design and test methodology that exploits the similarities in multiple asynchronous implementation styles through a general and unified abstraction, *Links and Joints*, and (2) demonstrate the embedding

of this methodology in a design flow — the systematic series of steps and activities involved in generating the circuit. This Link-Joint design and test methodology provides generality and uniformity of design and test interfaces. At the same time, designers enjoy flexibility with implementation decisions, design reuse, and more collaboration with other designers. The embedding in a design flow offers a structured route to scalable, accessible, and easy use of Links and Joints for designing small and large systems, from algorithmic programs to chips.

This dissertation gives key insights to circuit designers and distributed system designers about designing asynchronous circuits and systems with clear and well-defined interfaces that support design reuse and integrated testing. The embedding into a design flow showcases to EDA developers how to incorporate our methodology into their tools for more flexibility and generality in circuit representation.

1.1 Research Summary

The primary objective of this research is to *make it easy to insert asynchrony appropriate to each design part* by providing a framework for the design and test of asynchronous circuits, focused on using Links and Joints, a circuit-neutral model with built-in test and debug features. With Links and Joints providing clear boundaries and well-defined interfaces, we build a generic design and test flow to embrace multiple protocols and different asynchronous families.

Let's step back and consider what happens when designing asynchronous circuits. A key observation is that there are many decisions to make, such as handshaking protocol, signaling logic, data encoding, and circuit family. Many designers base some of these decisions on their familiarity and experience with certain options. In this dissertation, like many others in the past [2, 4, 5, 49, 72], our approach uses a language-based

transformation design flow — going from a high-level design description to custom circuits through a series of steps. Our approach features one key difference: the binding of implementation decisions is deferred as late as possible instead of making these decisions early. This difference is made possible by having Links and Joints as a mid-level abstraction in the design flow. The dissertation demonstrates our approach with design exploration and test emulation. It shows the generality and flexibility advantage of using our approach through the variety of protocols and asynchronous circuit families it supports and the ease of mixing circuit implementations.

1.1.1 Proposed Design Flow: *Ọnà*

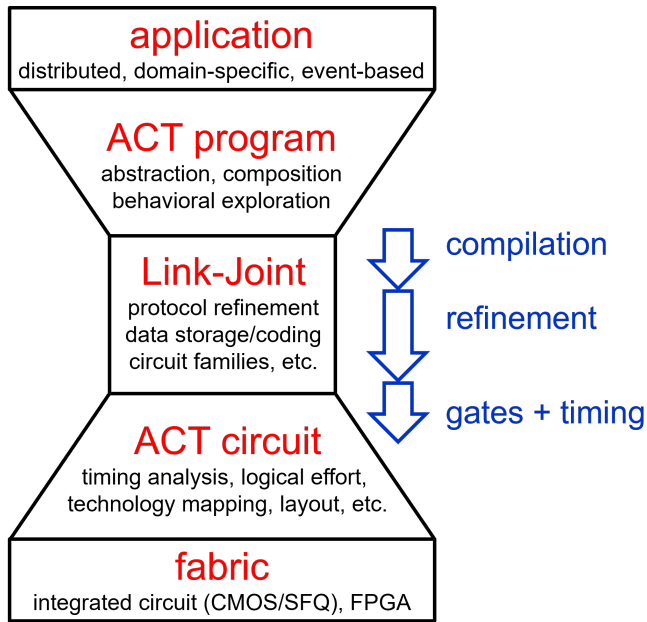


Figure 1.1: Our design flow, called *Ọnà* (/or-NUHR/, Yoruba for “way”).

Our design flow, *Ọnà* (Figure 1.1), centers around the use of Links and Joints as a methodology for the design and test of asynchronous systems. Links and Joints — introduced in detail in Chapter 3 — provide a way of thinking about asynchronous

systems in partitions of state (data storage) and action (computation and control). The state-action Link-Joint paradigm offers clear boundaries and well-defined interfaces for better embracing different asynchronous families by minimizing their differences and maximizing the similarities with an abstraction.

To facilitate the use of this methodology, we embed Links and Joints in an existing design flow instead of building a new one from scratch. We selected Yale’s *Asynchronous Circuit toolkit* (ACT), which is open source, in active use, and supports programs with data and control flow. We compile ACT programs into circuit-neutral Link-Joint networks, which we then translate into ACT circuits using stepwise refinement. We reuse ACT’s application and programming techniques at the top and ACT’s circuit and fabrication techniques at the bottom.

We consider testing an activity to be carried along in the design process. We approach testing with the same state-action separation for all levels of abstraction by defining observation structures and breakpoints. At the program level, we initialize and observe variables and set breakpoints in the program. At the Link-Joint level, we initialize and observe Link states and use a *go*-control (an external signal) to start and stop Joint actions. At the circuit level, we use *scan* access to initialize and observe Link states and use arbitrated *go*-controlled “*MrGO*” circuits to control actions in the circuits.

The uniformity in this test and debug approach gives us a basis to manage test complexity at the hardware. By planning our tests at a high level, we can reuse them — handing them down from one abstraction level to another by translating the observation and control structures to corresponding structures at the desired abstraction level. We apply this approach to structural and throughput tests.

1.1.2 Contributions

This research is a continuation of the Link and Joint research [60] started at the Asynchronous Research Center of Portland State University in 2015, particularly in making Links and Joints accessible to more people. While most of this dissertation is joint work with my supervisor and collaborators, the following key contributions are primarily mine.

- A semantics for Link-Joint networks using the shared variable model (presented in Chapter 3), the notion of *ports* as protocol-based interfaces between Links and Joints, and the abstract behavior implementation and validation of Link-Joint networks.
- The recognition that initialization determines structural protocol settings such as *active versus passive* and *push versus pull*, which rebuts the convention of dedicated communication ports per use case (Section 3.3).
- The adaption of the ACT compilation strategy for control and data flow programs to generate Link-Joint networks (presented in Chapter 4), including the development of target Link-Joint library elements.
- A set of refinement steps to illustrate the flexibility of stepwise application of circuit implementation decisions in the Link-Joint model, including the implementation and Verilog-based validation of Link-Joint network refinements for various choices in protocols, data encodings, storage requirements, and asynchronous circuit families (presented in Chapter 5).
- A unified test and debug strategy that can be translated from one abstraction level to another (presented in Chapter 6).

- A demonstration of Link-Joint networks as ACT formatted circuits with timing constraints (presented in Chapter 7) and mixed circuit implementations (presented in Chapter 8).

1.1.3 Publications Based on this Work

1. **E. Esimai** and M. Roncken, “Flexible Compilation and Refinement of Asynchronous Circuits,” 2023 28th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), Beijing, China, 2023, pp. 109-119, doi: 10.1109/ASYNC58294.2023.10239623.
2. M. Roncken, **E. Esimai**, V. Ramanathan, W. A. Hunt and I. Sutherland, “State Access for RSFQ Test and Analysis,” in IEEE Transactions on Applied Superconductivity, vol. 33, no. 5, pp. 1-7, Aug. 2023, Art no. 1303907, doi: 10.1109/TASC.2023.3251949.
3. **E. Esimai** and M. Roncken, “Flexible Active–Passive and Push–Pull Protocols,” in IEEE Embedded Systems Letters, vol. 14, no. 3, pp. 139-142, Sept. 2022, doi: 10.1109/LES.2022.3159492.
4. M. Roncken, I. Sutherland, and **E. Esimai**, “Micropipelines United,” in A. Brown and A. Yakovlev (eds) “We’re going to Need a Bigger Computer - Essays dedicated to Steve Furber on the occasion of his retirement. At Last,” University of Manchester Press Unit, 12 January 2024.
5. ASYNC 2022 Summer School: 3-day online seminar given by Rajit Manohar (Yale University), Benjamin Hill (Intel), Montek Singh (University of North Carolina at Chapel Hill), and Marly Roncken, **Ebele Esimai**, and Ivan Sutherland (Portland State University). Audio presentations and slides are available

at <https://asyncsymposium.org/async2022>. The Portland State presentations cover (1) Links and Joints: behavioral design and (2) Links and Joints: gate-level design

1.2 Dissertation Organization

Chapter 2 introduces fundamental concepts of asynchronous circuit design, such as handshaking, signaling, data encoding, and asynchronous circuit families. It surveys some examples of design flows and covers testing techniques. This chapter builds the foundation for understanding asynchronous circuit design and testing.

Chapter 3 discusses the circuit-neutral Link-Joint model in detail as an abstraction of asynchronous (self-timed) systems. This discussion includes the Link-Joint structure, its semantics, and the Link-Joint simulation model in Verilog. This chapter serves as an introduction for readers unfamiliar with Links and Joints.

Chapter 4 introduces *Compilation* as a crucial component of our design flow, *Qnà*. It presents a front-end solution for Links and Joints where late binding of implementation decisions is made possible using syntax-directed translation by compiling high-level programming language designs into a network of Links and Joints. The chapter introduces the languages in the source programs and the target Link-Joint library elements. The chapter also demonstrates the compilation scheme.

Chapter 5 introduces another crucial component in our design flow: *Refinement*. The chapter discusses the stepwise decisions for choosing implementation bindings. The chapter showcases Link-Joint network refinement examples and some abstract gate-level implementations.

Chapter 6 focuses on test, debug, initialization, and simulation. It highlights the significance of testing and debugging in the design process, concentrating on

translating test sequences and connecting test and debug at all abstraction levels in the design flow. The chapter showcases both functional and structural tests and covers test simulation tools like Verilog and ACTSIM.

Chapter 7 discusses getting back into the ACT ecosystem to complete the shallow embedding of Links and Joints into ACT. The chapter demonstrates how we specify an abstract Link-Joint gate-level implementation in the ACT format, with Product Rule Set and timing constraints.

Chapter 8 discusses the ease of mixing and matching circuit implementations using the Link-Joint model.

Chapter 9 concludes the work by summarizing the achievements and limitations. It also provides an outlook on future enhancements and activities.

Chapter 2: Background

Digital circuit design comes in two significant categories — *synchronous* and *asynchronous*. A circuit is synchronous when all its components share a common and discrete notion of time, as defined by a “clock” signal distributed throughout the circuit [66]. Otherwise, a circuit is asynchronous (or self-timed) when it has no global clock. An asynchronous circuit uses local protocols between its components to govern communication, synchronization, and sequencing operations [66]. Asynchrony offers benefits over synchrony in key metrics such as modularity and composability, high performance, low peak power, low energy dissipation, low electro-magnetic emission noise, and lack of global timing and global clock skew and distribution problems, especially for variable and data-dependent computations [71].

This chapter provides some background on the key components of asynchronous design and testing. The chapter covers widely used communication protocols, signaling logics, data encodings, and asynchronous circuit families. However, we do not prescribe which of them to use, nor do we indicate a preference for one over the other in this dissertation. Rather, we show that our design flow supports implementations that use, mix, and match the protocols, signaling logics, data encodings, and asynchronous families discussed in this chapter. We showcase these implementations in Chapters 5, 6, and 8. Also, the chapter introduces some existing approaches and design flows for asynchronous design related to the work done in this dissertation. After that, we briefly outline the test and debug basics and techniques for asynchronous circuits in preparation for Chapter 6.

2.1 Communication Protocols

Asynchronous circuits operate via local communication between their components. Because they have no global clock to sequence their operations periodically, asynchronous circuits require a protocol, typically referred to as *handshake protocol* or simply *handshake*, to indicate readiness to communicate and send or receive data and the validity of data [66].

2.1.1 Handshakes

Local communication in asynchronous circuits uses *handshakes* to govern the operations of communicating neighbors. Typically, one neighbor or communication side is dedicated as an *initiator* and starts the communication and data transmission with a request event. The other side is dedicated as a *responder* and signals the receipt of communication and data transmission with an acknowledgment event. Toggling between initiator and responder events forms a *handshake protocol* between neighboring components [14].

Traditionally, the neighbors playing the roles of *initiator* and *responder* are fixed. However, in Section 3.3, we will show that the roles do not have to be fixed but are dependent on how the circuit is initialized and that roles can be initialized in multiple ways depending on the execution purpose — for normal execution or test and debug.

The handshake protocol can be implemented in several forms. The most typical implementation uses two signals, *request* and *acknowledge*, with an explicit initiator and responder. *Request* indicates the start of an action, while *acknowledge* indicates the completion of an action. We use this implementation to describe two commonly used handshake protocols: a 2-phase handshake and a 4-phase handshake.

2.1.1.1 2-phase Handshake

The **2-phase handshake** has two consecutive execution phases. In the first phase, the *initiator* starts the handshake protocol by changing the level of the *request* signal. In the second phase, the *responder* completes the protocol with a corresponding level change on the *acknowledge* signal.

For example, in Figure 2.1, the *sender* starts the first phase and initiates the protocol by sending valid data to the receiver and by notifying the receiver that the data are valid via raising the *request* signal. This allows the receiver to compute on the valid data. When done computing, the *receiver* starts the second phase and completes the protocol by raising the *acknowledge* signal, which tells the sender that it may change the data and start the next communication when it has valid data for the receiver again. Note that the next (second) communication in Figure 2.1 lowers the *request* signal in its first phase and then lowers the *acknowledge* signal in its second phase.

A 2-phase handshake with separate *request* and *acknowledge* signals takes two handshakes to get back to the same *request-acknowledge* state values that it started with. Because many designers start the protocol with *request* and *acknowledge* both low, a 2-phase handshake is often referred to as a non-return-to-zero (Non-RTZ)

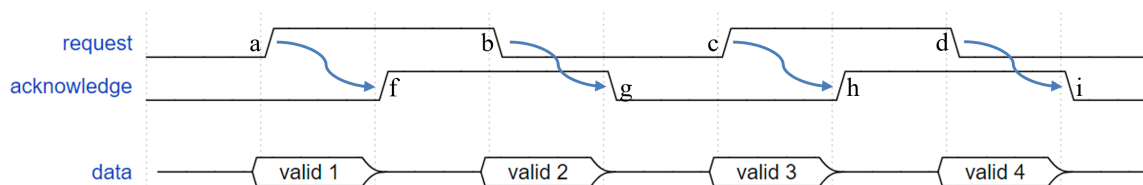


Figure 2.1: A typical 2-phase handshake protocol with *request* and *acknowledge* signals. The waveforms show four successive 2-phase handshakes: $a \rightarrow f$, $b \rightarrow g$, $c \rightarrow h$, and $d \rightarrow i$. Note that each handshake has its own valid data. The data are valid from when the handshake starts to when it finishes but can be changed from when it finishes to when the next handshake starts.

protocol. We will adopt this term even though our design approach allows a 2-phase protocol to start in any of four possible states, i.e., with *request-acknowledge* both low, both high, low-high, or high-low.

2.1.1.2 4-phase Handshake

The **4-phase protocol** has four execution phases, executing two successive 2-phase handshakes — the first to set the request and acknowledge signal and the second to reset the signals to their initial values. Because designers typically start each 4-phase handshake with low *request* and *acknowledge* signals, a 4-phase handshake is often referred to as a return-to-zero (RTZ) protocol. We will adopt this term. Figure 2.2 (top) shows two consecutive 4-phase handshake protocols, $a \rightarrow f \rightarrow b \rightarrow g$, and $c \rightarrow h \rightarrow d \rightarrow i$.

A 4-phase handshake allows more than one data validity scheme. In Figure 2.2, the top data waveform shows when data remain valid using bundled data encoding, discussed in Section 2.1.3. The bottom data waveform shows data that remain valid in the first and second phases and are reset to NULL in the third phase. The data become valid again sometime in the fourth phase, at the latest (as shown here) at the

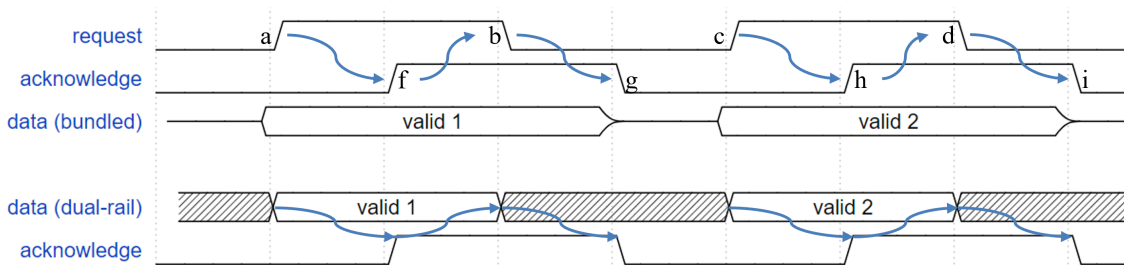


Figure 2.2: A typical 4-phase handshake protocol with request and acknowledge signals with bundled data (top) and a typical 4-phase handshake protocol with dual-rail data and acknowledge signal (bottom). Section 2.1.3 discusses the different data encodings. Note that each handshake has its own valid data.

start of the next first phase. The bottom data scheme is typical for 4-phase dual-rail encoded data, discussed in Section 2.1.3.

2.1.2 Signaling Logic

Signaling logic refers to how the local communication control signals are interpreted. Options include **level logic**, **transition logic**, and **pulse logic**, which we differentiate as follows.

- **Level logic** [66] interprets signals as Boolean levels — 0 or low, and 1 or high. Level signaling is the most commonly used signaling logic in both synchronous and asynchronous circuits and systems. Figures 2.1 and 2.2 show 2- and 4-phase handshake protocols for level signals *request* and *acknowledge*.
- **Transition logic** [65, 69] interprets signal changes (transitions). With Boolean logic, we can view transition logic as generating a rising signal transition from 0 (low) to 1 (high) or a falling signal transition from 1 to 0. Note that a 2-phase signal transitioning protocol looks very much like the 2-phase protocol in Figure 2.1 at the interface between the initiator-sender and responder-receiver. However, there is much more to the logic that generates the behavior at the interface. The logic based on level changes (transitions) is very different than logic based on levels, as we will show in Chapter 5.
- **Pulse logic** [27, 44, 51, 57] interprets signal pulses. With Boolean logic, we can view pulse logic as generating 0(low)-to-1(high)-to-0(low) or 1(high)-to-0(low)-to-1(high) pulses. We use pulse logic to describe superconducting rapid single flux quantum (RSFQ) [28] computations by encoding and interpreting the presence of a pulse on a signal as 1 or *TRUE* or *high* and the absence of a

pulse on a signal as 0 or *FALSE* or *low* [27,57]. We cover more details on pulse logic in Chapter 6. Figure 2.3 shows a 2-phase handshake protocol for pulse signals *request* and *acknowledge*.

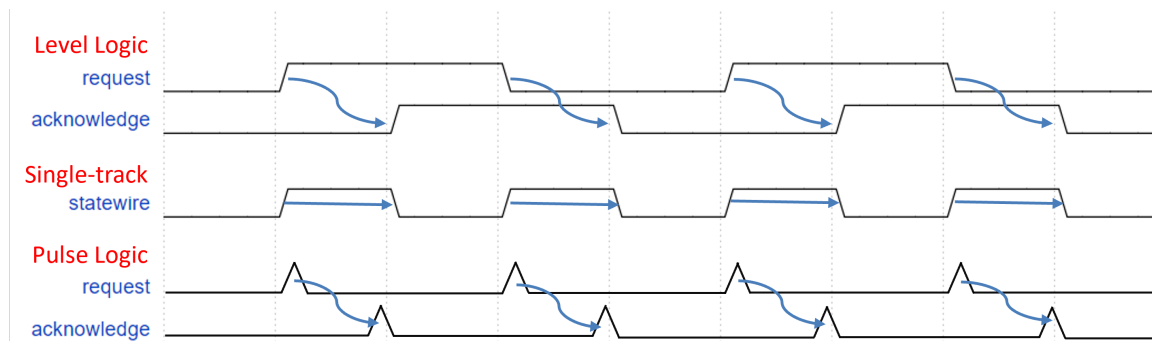


Figure 2.3: 2-phase handshake protocol with two separate level signals, *request* and *acknowledge* signals, as in Figure 2.1 (top) set off against a single *statewire* signal for single-track circuit families such as GasP (middle) and against two separate pulse logic *request* and *acknowledge* signals for superconducting circuit families such as RSFQ (bottom).

2.1.3 Data Encodings

Data encoding refers to the representation of data signals. Common choices are **single-rail (bundled data)** and **dual-rail** data encodings [66].

Bundled data encoding uses one wire for each bit of information. The sender transfers an n -bit data value to the receiver on n wires. The logical level of the data signal represents either a logic 1 or a logic 0. Bundled data is popular due to the ability to separate control from datapath [2]. The bundled data protocol is simple and practical, providing lower power and energy dissipation compared to dual-rail data encoding.

Dual-rail data encoding uses two wires per data bit of information. An n -bit data value is transferred on $2n$ wires plus the signaling wires for the protocol control

logic. The data bits can be used to generate the *request* signal or generate the first handshake phase in the case of single-track. In other words, part of the handshake comes from the data encoding. This protocol is delay-insensitive. A typical dual-rail encoding has four states: (00) Idle (data are not valid); (10) Valid 0-bit; (01) Valid 1-bit; and (11) Illegal. Dual-rail encoding is insensitive to delays on any wire but has increased complexity in both wiring and logic [14]. Dual-rail encoding is an example of an N-of-M encoding scheme where $N=1$ and $M=2$. N-of-M encodings use groups of M wires to encode data values by considering data valid as soon as N wires are activated [66].

2.2 Asynchronous Circuit Families

There are many circuit implementation styles. A circuit family is a collection of circuit implementations optimized for a particular goal.

Click [50] developed at Philips in Eindhoven, the Netherlands, is among the most popular asynchronous circuit families used today. The family uses edge-triggered D flip-flops and avoids using other state-holding circuit elements for maximum compatibility with traditional synchronous EDA tools, particularly those for test and timing analysis [66]. The original version uses 2-phase level-signaling bundled-data protocols, but we implement a 4-phase version as well in Chapter 5.

GasP by Sun Microsystems Laboratories [68] is another level-signaling asynchronous circuit family. *GasP* uses single-track signaling logic (a statewire). It was originally designed for 2-phase handshakes, but we also implement a 4-phase version in Chapter 5.

The *Set-Reset (SR)* family is a simplified version of *GasP* that uses a shared statewire but drives it with set-reset latches instead of with drive-high-keep-low and

drive-low-keep-high *GasP* latches [60].

Ivan Sutherland developed *Micropipelines* [69], an efficient 2-phase bundled data asynchronous pipeline style. This bundled-data asynchronous pipeline style uses transition signaling in its control and uses double so-called *capture-pass* latches in its datapath [66, 69].

Mousetrap by Singh and Nowick [65] is an alternative transition signaling implementation for Micropipelines that use single latches [63]. It is an elegant implementation optimized for high speed in that the latches are normally transparent; i.e., the data latches begin transparent and become opaque just after new data arrive [20].

Level and transition signaling families such as *Click*, *GasP*, *Set-Reset*, *Mousetrap*, and *Micropipelines* have known implementations in complementary metal oxide semiconductor or CMOS manufacturing technologies. Alternative superconducting manufacturing technologies, such as *Rapid Single Flux Quantum (RSFQ)* [27], can be used for either synchronous or asynchronous circuits. Asynchronous implementations require a pulse-logic and either bundled or dual-rail pulse-logic data encodings [31, 57]. Superconducting circuit families provide benefits for very fast operating speeds and low energy consumption.

This dissertation continues the unified abstraction research for the various asynchronous circuit families [60] presented in 2015. In Chapter 3, we develop a model for the abstraction, and in Chapters 5, 6, and 8, we extend the implementation to other families. Our abstract Link-Joint interface gives a “start-finish” view of the various protocols shown in Figures 2.1, 2.2 and 2.3, which results in the same abstract view for 2-phase and 4-phase protocols. We use step-wise refinements to create different implementations, such as 2-phase versus 4-phase protocols. These refinements are discussed in Chapter 5.

2.3 Design Approaches

There are a couple of design approaches for large and scalable asynchronous circuits. These approaches include *classical or burst-mode asynchronous state machines* [14, 66], *Graph-based specification*, such as Signal Transition Graphs (STG) [9], *Desynchronization* and *Compilation*. *Desynchronization* is a methodology of “deriving asynchronous circuits from optimized synchronous circuits by replacing the clock distribution tree by a handshaking network” [11, 12]. This methodology aids the adoption of asynchronous circuits from the familiar base of synchronous circuits and commercial EDA tools. There are many applications and design flows using this methodology [30, 45, 75, 78, 81].

The *Compilation* approach involves some form of transformation of a high-level specification, usually a program [72]. The high-level specification could be language-based, such as using Communicating Sequential Processes (CSP) [23], OCCAM [29], or Communicating Hardware Processes (CHP) [42]. For distributed and parallel systems, a designer would program both the functional and communication behavior of the system and then, through automated and possibly interactively guided steps, generate a circuit that satisfies this program. Typically, the generated circuit uses a gate-level cell library and gate-level modules. The modules can be generated in advance and can be optimized in situ, using peephole optimization or dedicated synthesis approaches [2, 4, 13, 49, 52, 74].

This dissertation focuses on using a compilation-based design flow but leaves open the opportunity for local low-level synthesis.

2.3.1 Compilation-Based Design Flows

Compilation-based syntax-directed design flows typically capture high-level specifications in an algorithmic programming language like Communicating Sequential Processes (CSP) [23], with primitives for sequential and parallel processing of computation, non-deterministic and deterministic selection, and communication for synchronization and message passing. Syntax-directed compilation [6] gives the designer complete control over the implementation: *you get what you program*, not just behaviorally but also structurally and topographically. The structures in the programs map one-to-one to the structures in the implementation.

Tangram by Philips [74] (which evolved into Haste by Handshake Solutions) is one of the early syntax-directed asynchronous compilation languages. To emphasize the tight connection between Tangram programs and asynchronous circuits, the team dubbed their approach *VLSI programming*. Tangram programs are compiled into intermediary representations, called *Handshake Circuits*, which, depending on the compiler settings, are mapped to specific circuit modules; that is, circuit implementation decisions are per design. Despite their success, Philips and Handshake Solutions discontinued and shelved Tangram and Haste and their related compilers and libraries between 2010 and 2012.

Another syntax-directed design flow is *Balsa* [2]. Balsa was developed at the University of Manchester and is similar to Tangram. It was created to provide an open-source alternative to Tangram. It offered a slightly different approach to handshake circuits by promoting the use of components parameterized by their behavior, terminal number, and size [3]. Balsa's development and management stopped in 2010.

Around the same time Philips started Tangram, Caltech developed an asynchronous syntax-directed design flow from a programming language called *Communi-*

ating Hardware Processes (CHP) [39,40,42], similar to, but not the same as *Tangram*. The key difference is that CHP programs can *probe* communications. For example, a communication partner can check if the other side is ready to communicate and make decisions based on that information. In contrast, CSP and *Tangram* can make communication decisions only when both communication partners are ready. CHP uses probes in guards of selection statements [34]. Manohar and Ataei [1] continued the use of CHP by developing an open-source EDA flow for asynchronous logic named *Asynchronous Circuit Toolkit (ACT)*. ACT is the only surviving general-purpose syntax-directed design approach for asynchronous circuits available today.

Erik Brunvand [4] also had a design flow focused on dataflow applications. The source language for this flow was OCCAM, written in LISP style. The design flow also used syntax-directed translation and implemented 2-phase bundled-data transition signaling circuits.

Handshake circuits are an intermediate representation — a network of handshake channels and components [74]. At the handshake circuit level, a designer picks communication protocol, data encoding, and circuit family to be implemented for the entire design. This representation is then used to derive the final circuit netlist, which goes into a CAD framework for placement, routing, simulation, and verification. Figure 2.4 shows the place of handshake circuits in the *Tangram* design flow.

Tangram, CHP Caltech, and ACT all started with asynchronous circuits that use 4-phase protocols for level signaling and quasi-delay-insensitive (QDI) data encodings like dual-rail data circuits [39, 41, 72, 73]. QDI circuits operate correctly regardless of the delays on their gates and wires but require “isochronic forks” — forked wires where all branches have exactly the same delay [14]. Because data results indicate the completion of a data computation, QDI circuits can easily exploit average delay advantages and thus produce circuits with higher throughput and lower latency. Balsa

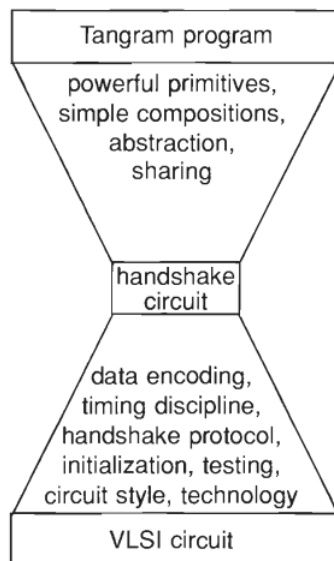


Figure 2.4: Handshake circuits: Intermediary in the Tangram design flow. (Taken from Handshake circuits: an intermediary between communicating processes and VLSI, Figure 0.5, page 13 [72].)

also implemented 4-phase bundled data protocol circuits [2]. When power became an issue in circuit design rather than mere speed, the asynchronous community started looking at 2-phase protocols with single-rail or bundled data. Tangram was one of the earlier adopters [49].

A vital observation of these design flows is that circuit implementation decisions (which handshake protocol, signaling, encoding, and family to use) are made and fixed for each design prior to compilation. Also, handshake circuits have a fixed notion of port protocols – active, passive, push, and pull. In other words, the implementation library has distinct implementations for each protocol choice, hence tying the implementation closely to the representation.

Following the example of the design flows in this section, this dissertation uses a compiler-based, syntax-directed design flow. However, we focus on flexibility available through the Link-Joint model with clear boundaries and well-defined interfaces for finer granularity of circuit implementation decisions as opposed to global imple-

mentation decisions applied to the entire design. We delay implementation decisions as late as possible to support mixed implementation of multiple protocols, signalings, data encodings, and families. We provide more details in Chapters 3, 4, 5, 6, and 8.

2.4 Test and Debug

Many testing techniques of asynchronous circuits focus on validating if the fabricated circuit has any physical faults [25, 53–56, 79, 80] because formal verification techniques go only as far as the circuit layout masks submitted for manufacturing. Manufacturing defects can be discovered only by testing the fabricated circuits. Because functional tests may not, and generally will not, provide enough coverage of manufacturing defects within a reasonable test time, manufacturing tests are typically based on so-called *structural fault models* for manufacturing defects.

Fault models simplify test generation by abstracting from actual physical defects or environmental influences that result in malfunctions of a circuit [79]. The term *structural testing* refers to testing based on structural fault models. A well-known structural fault model is the *stuck-at fault* model that assumes that a signal, gate input, or gate output remains (is stuck at) 0 or remains (is stuck at) 1. Other standard fault models for asynchronous circuits include functional and delay fault models.

Asynchronous circuits are mainly autonomous and provide minimal opportunities to pause system operations, whether for observing the current state contents or altering the state contents [53, 56]. Design-for-testability (DfT) techniques increase circuits' testability by adding test circuitry during design. These techniques improve opportunities for *controllability* and *observability*. Scan test, a powerful DfT technique, addresses different fault models by shifting in and out test patterns generated

for a specific fault model.

Roncken et al. built initialization, test, and debug into the Link-Joint model [43, 60]. This feature makes the Link-Joint model both a design and test model. With an emphasis on the separation of state and action, they brought the necessary requirement for testing asynchronous circuits into modeling, thereby addressing the needs of controllability, observability, and non-determinism. They introduced a new circuit element, “*MrGO*”, to safely pause or start individual actions and used scan access to observe or alter state.

In this dissertation, we harmonize test and debug throughout the design flow by connecting DfT, test, and debug mechanisms at all levels of abstraction in our design flow to have a unified test approach. Chapter 6 exhibits our implementation of a test approach that starts at a high level and flows down with compilation and refinement to lower design levels.

Chapter 3: Links and Joints

First presented at the IEEE International Symposium on Asynchronous Circuits and Systems in 2015 by the PSU Asynchronous Research Center, the Link-Joint model [60] provides a *circuit-neutral model with integrated test and debug*. Links and Joints were created as an abstraction of self-timed systems. Regardless of the many possible circuit family implementations a design may use, the Link-Joint model keeps their implementation differences “under the hood” [60–62]. The paradigm facilitates a way of thinking about self-timed circuits [62] by partitioning systems into:

- transport and storage of data and other state information (Links), and
- computation and flow control actions (Joints)

This chapter is intended as an introduction for readers unfamiliar with Links and Joints. The chapter covers the Link-Joint structure, its inherent features for test and debug, its semantics, and its simulation model.

3.1 The Link-Joint Model

The idea of Links and Joints started when its creators began working with two different circuit families [61]. They found enough similarities in various parts of the implementation that resulted in duplicated effort for each implementation. Typically, most abstractions for circuits have all the functionality grouped together and have connections for handshake communication between the functional units [66]. Links and Joints shift the view of circuit abstraction by moving the interface boundary such that handshake communication and state, as well as the different ways they can be

implemented, become internal to *Links*, while computation and flow control become internal to *Joints* [60–62]. This circuit abstraction view results in a network with Joints as nodes and Links as edges connecting the nodes.

Consequently, even without considering the circuit implementation, we can view an abstraction for adding two values as a Link-Joint network shown in Figure 3.1. We express this computation in a data-driven way, where the arrival of data values initiates the computation action. By separating the state from computation, the values to be added and the computation result are stored in Links, marked as rectangles, connected to the addition computation unit, Joint, marked as a big circle.

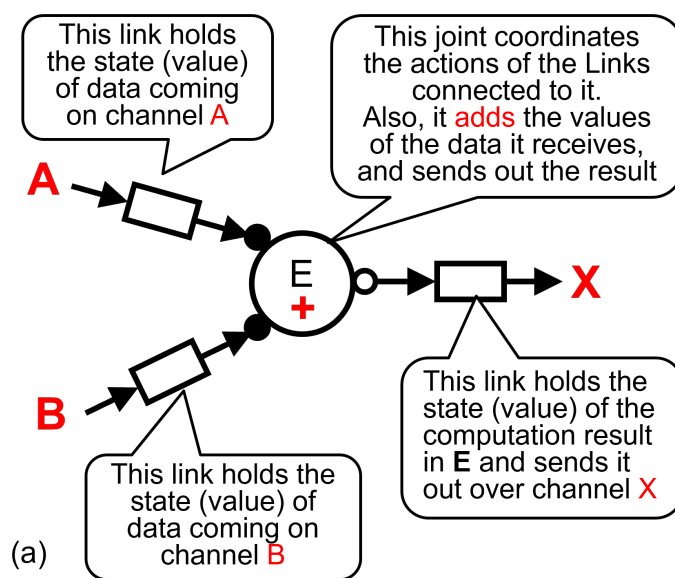


Figure 3.1: A simple Link-Joint network that adds two numbers as a data-flow computation with channels, $A + B \rightarrow X$.

We can also express the addition computation in Figure 3.1 in a control-driven way, where the functional units control the pace of actions in the circuit. The same state-action separation also works with this expression, shown in Figure 3.2. Chapter 4 further discusses different ways of specifying asynchronous circuits. Note that Joints have different types based on their purpose. For example, Figure 3.2 shows three

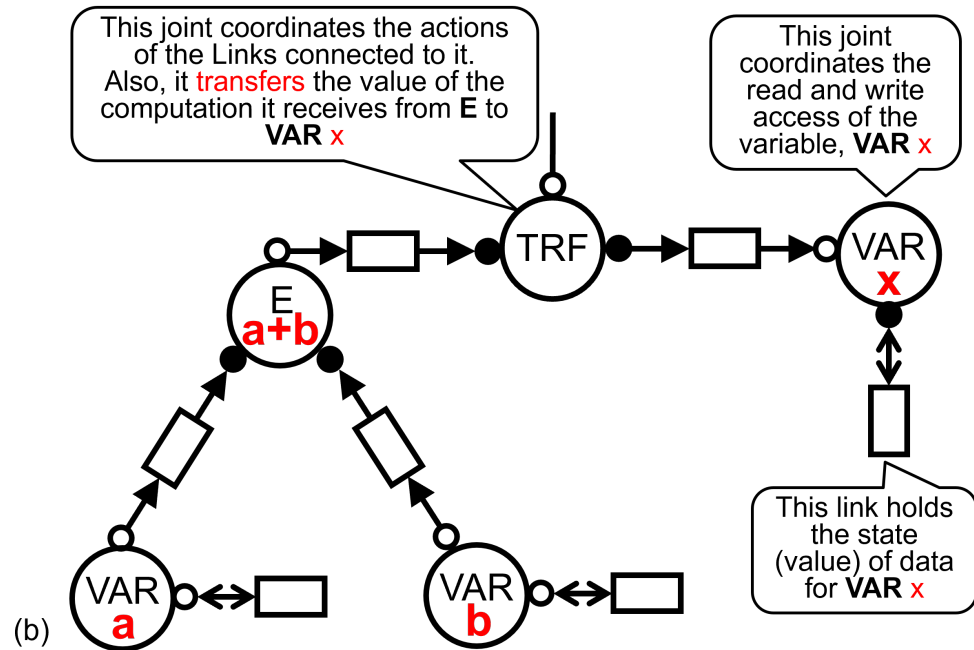


Figure 3.2: A simple Link-Joint network that adds two numbers as a control-flow computation with variables, $x := a + b$.

different types of Joints: **E** for expression evaluation, **TRF** for data transfer, and **VAR** for read or write access to the stored value for a variable. More Joint types are introduced and discussed in Section 4.2.

Therefore, Links transport and store state information. Joints are stateless. They compute and control the flow of information between the Links to which they connect. Joints may connect to multiple Links. Each Joint acts based on the state of its connected Links. *In essence, each Joint is a place where Links meet to exchange information.* A Link-Joint network alternates Links and Joints by interfacing them using *port* connections. The interfacing ports are marked as small circles colored either white or black in Figures 3.1 and 3.2.

The Links form a distributed state space because there is a clear separation between state storage and action control. In the absence of a global clock to synchronize actions, local state changes serve as actionable events and trigger the flow of control

locally. Actions are atomic, local, and spread over Joints. State-action separation also offers an inherently intuitive approach to test and debug. Each Joint has its own (arbitrated) *go* signal [60] – an external signal to start or stop any or all Joint actions. *Go* signals make it possible to control and observe any or all Link states *externally* without any Joint action interference. The presence of *go* signals simplifies and improves initialization, test and debug. Chapter 6 connects these state-action control and observation features in Links and Joints to observation and control structures for design initialization, test, and debug in both programs and circuits.

The Link-Joint model is *circuit-neutral*; it is silent about the implementation style (protocols and circuit families) used in the final circuit. The Link-Joint model is flexible because it decouples itself from the final circuit implementation. Implementation is entirely dependent on design choices [16]. The Link-Joint model provides an abstraction level between algorithmic programs and electronic circuits when embedded in a design flow [16, 17, 58, 59]. The Link-Joint model allows for experimenting with different implementation options and binding design choices as late as possible. In addition, this flexible design approach facilitates the easy mixing of protocols and families and promotes design reuse.

3.2 Shared Variable Semantics

In early Link-Joint publications, the model for the Link has always been implicit [60–62]. In 2022, we introduced a shared variable semantics that, for the first time, models a Link explicitly [16]. Note that the shared variable semantics is not completely formalized; the semantics provide intuitive definitions for Links, Joints, and well-formed Link-Joint networks.

A Link is a structure with two distinct *ports* with formal names *A* and *B*. A Link

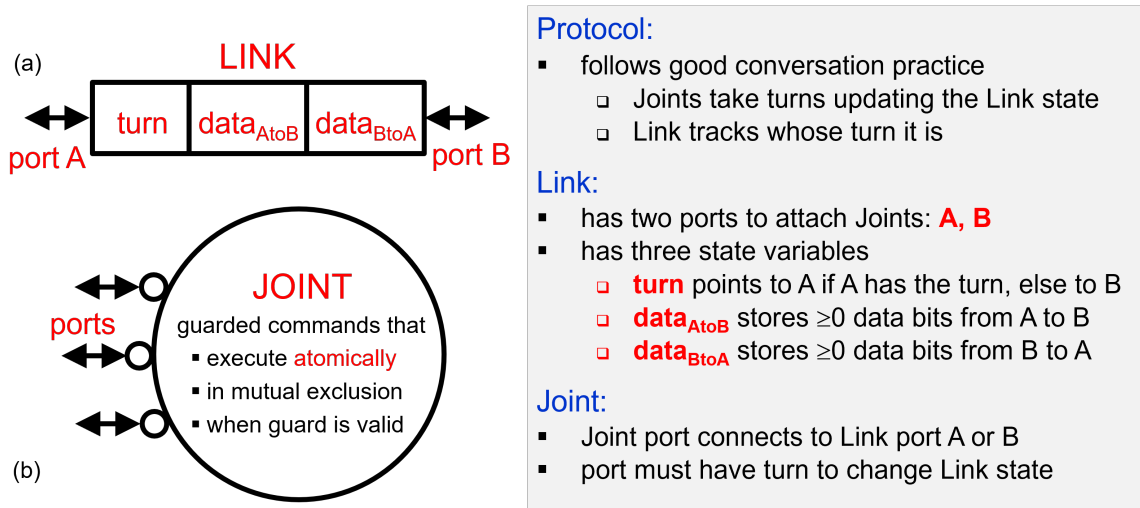


Figure 3.3: Protocol and model of a Link (a) and a Joint (b).

has three state variables, $turn$, $data_{AtoB}$, and $data_{BtoA}$ (Figure 3.3(a)), which is shared typically by exactly two Joints connected at Link port A and Link port B respectively. The atypical case is when an *environment* is at one of the Link ports instead of a Joint. A Joint solely acts on the states of its connected Links. It has ports, each of which connects to either a Link port A or B (Figure 3.3(b)).

Following good conversation practice, the two Joint ports connected to opposite ends of the same Link take turns updating the Link variables, including variable $turn$. For each Link, its variable $turn$ designates which of the two ports, A or B , may update the Link variables. When $turn$ designates A , denoted as " $turn \equiv A$ ", port A — ultimately representing a Joint or environment connected to A — may update $turn$ and $data_{AtoB}$. Specifically, port A may update $data_{AtoB}$ with new data going from A to B and may change $turn$ to designate B . Likewise, when " $turn \equiv B$ ", port B may update $turn$ and $data_{BtoA}$. Link variables $data_{AtoB}$ and $data_{BtoA}$ contain zero or more bits of data going from A to B and B to A , respectively. Therefore, a Link can be bidirectional, unidirectional, or dataless depending on the number of bits in $data_{AtoB}$ and $data_{BtoA}$.

Conforming with the earlier Link-Joint publications [60–62], the shared variable semantics specifies Joint behaviors as guarded commands [15]. Each guarded command of a Joint is of the form $guard \rightarrow command$, where $guard$ is a Boolean expression based on the variables of its connected Links and $command$ a sequence of assignments changing the variables of its connected Links. For integrated test and debug, a Joint’s go signal is included in its guarded command, such that go permits and $\neg go$ prohibits command execution.

Joint specifications use the following terms to access a connected Link l ’s variables from a Joint port p , where p connects to either Link port $l.A$ or Link port $l.B$.

- $myturn(p)$: A Boolean that indicates if it is p ’s turn to update the shared variables in Link l . For example, if Link port $l.B$ connects to the Joint at port p and “ $l.turn \equiv B$ ”, then $myturn(p)$ is *TRUE*.
- $yourturn(p)$: An assignment that relinquishes p ’s turn on connected Link l to the port at the other end of Link l , p_{peer} . This assignment makes $myturn(p)$ become *FALSE* and $myturn(p_{peer})$ become *TRUE*. For example, if Link port $l.B$ connects to the Joint at port p , and Link port $l.A$ connects to the Joint at the other end of the Link at port p_{peer} , then $yourturn(p)$ sets “ $l.turn \equiv A$ ”, making $myturn(p)$ *FALSE* and $myturn(p_{peer})$ *TRUE*.
- $myR(p)$: Data read by port p (data going out of Link l ’s data variable through Joint port p). For example, if Link port $l.B$ connects to Joint port p , then $myR(p)$ refers to data stored in Link variable $l.data_{AtoB}$ or if Link port $l.A$ connects to Joint port p , then $myR(p)$ refers to data stored in Link variable $l.data_{BtoA}$.
- $myW(p)$: Data written by port p (data going into Link l ’s data variable through

Joint port p). For example, if Link port $l.B$ connects to Joint port p , then $myW(p)$ refers to data written into Link variable $l.data_{BtoA}$ or if Link port $l.A$ connects to Joint port p , then $myW(p)$ refers to data written into Link variable $l.data_{AtoB}$.

- *go*: Boolean signal set externally by the environment. When TRUE, the Joint has permission to act. When FALSE, Joint action is prohibited.
- For multiple Joint ports p_1 and p_2 , we use abbreviations $myturn(p_1, p_2)$ for $myturn(p_1) \wedge myturn(p_2)$, and $yourturn(p_1, p_2)$ for $yourturn(p_1); yourturn(p_2)$ — where “;” indicates that the assignments are in sequence.

This terminology allows the specifications to be silent as to whether Joint port p connects to Link port A or Link port B . Joints can have multiple guarded commands. We may use $guard_1 \rightarrow guard_2 \rightarrow command$ as an alternative notation for $guard_1 \wedge guard_2 \rightarrow command$. Guarded commands execute atomically, in mutual exclusion, and only when their guard is valid [15].

A Link-Joint network is a collection of connected and alternating Link and Joint instances. If a Link port is not connected to a Joint or a Joint port is not connected to a Link, it is connected to the environment.

For example, Figure 3.4(a) shows a tiny Link-Joint network with two Links, $L1$ and $L2$, and Joint $COPY$, $J1$. The (single) guarded command of Joint $COPY$ in Figure 3.4(b) specifies that when port *in* has the *turn* on Link $L1$ and port *out* has the *turn* on Link $L2$, and the Joint has *go* permission, then the data read from Link $L1$, $myR(in)$, can be written to Link $L2$, $myW(out)$. After such a write operation, port *in* relinquishes its *turn* on Link $L1$, and port *out* relinquishes its *turn* on Link $L2$.

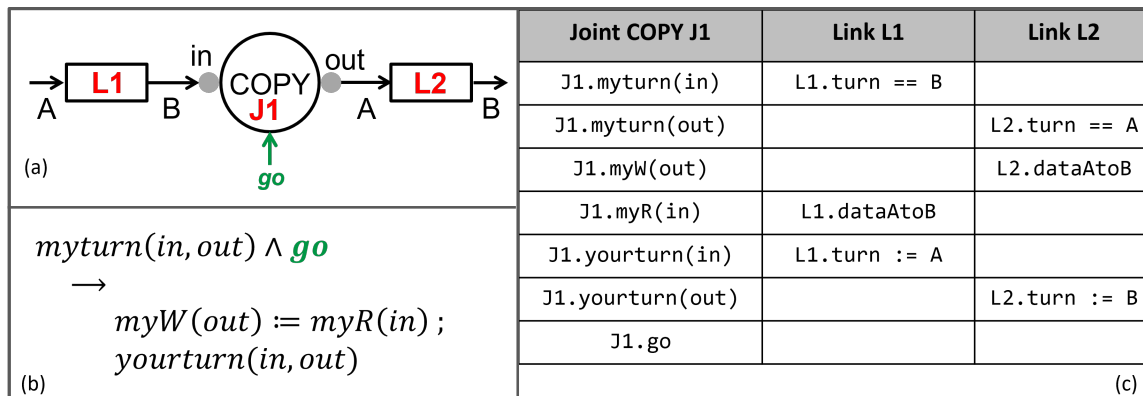


Figure 3.4: Example Link-Joint network with two Links, $L1$ and $L2$, and one Joint $COPY$ (a), with a guarded command specification for the Joint (b), and an example interpretation of the Joint's terms in relation to its connected Links (c). Note that each Link port is marked with a small circle, colored grey, to indicate that who initially has the *turn* on either Link $L1$ or $L2$ is not yet set.

3.2.1 Semantics in Action

To illustrate the protocol of Links and Joints, we use a simple Link-Joint network with a data-less Link and two $SKIP$ Joints; we call it **Ping-Pong** (Figure 3.5). The guarded command specification of $SKIP$ is included. $SKIP$ has one port p , does not do any data operations, and when it has both the *turn* on the Link connected at port p and *go*-permission, it acts by giving the turn to whoever is on the other end of the

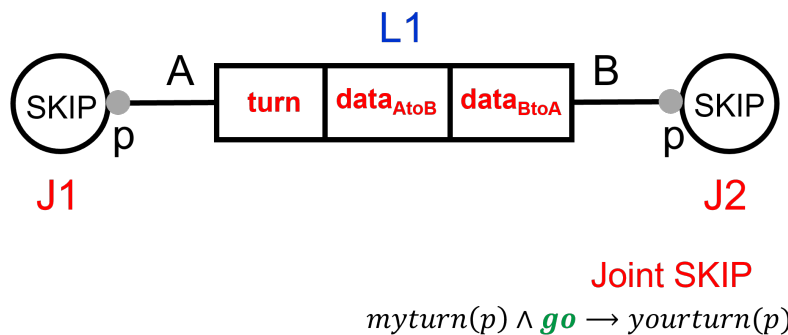


Figure 3.5: Ping-Pong Link-Joint network with a Link and two $SKIP$ Joints with the guarded command specification of Joint $SKIP$.

connected Link (a **no-op** operation). We show the use of the Link states and the *go* signals in Joints to initialize the design, run it for a while, and then stop it.

In Figure 3.6, we initialize the network. First, we disable each Joint. Each Joint has an external signal, *go*, to permit or prohibit Joint actions. To prohibit Joint actions, we make each *go* signal **FALSE** as shown in Figure 3.6. Then, we initialize the *turn* variable of the Link, which is possible because the Link-Joint model assumes that we have external access to the Link variables.

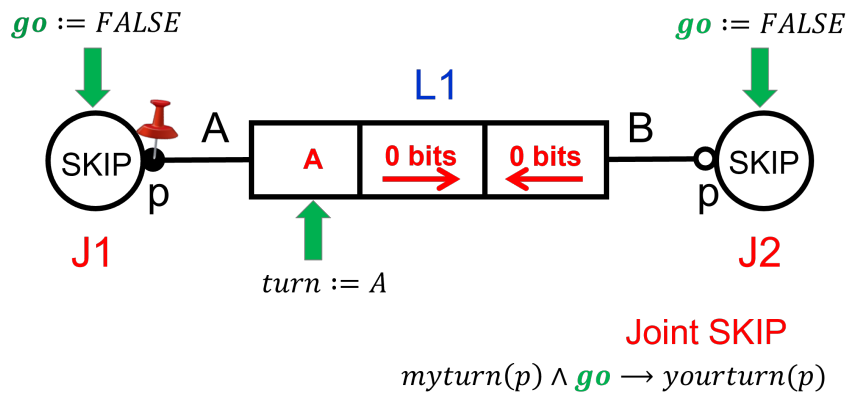


Figure 3.6: Initializing the Ping-Pong Link-Joint network.

We set the *turn* to port *A*, making the Joint at port *A* act first when given *go*-permission. We mark this selection by coloring port *A* black, indicating that port *A* has the initial turn, and by coloring port *B* white, indicating that port *B* does not have the initial turn. In addition, we added a red-colored pin to mark where the turn is during the execution of PingPong. The turn is where the pin is. The two data variables in the Link have zero bits. Therefore, Link *L1* is a data-less Link.

After initialization, we need to enable the Joints to make the network run. Suppose we give Joint *J2* *go*-permission ($J2.go := TRUE$) but not Joint *J1* ($J1.go := FALSE$), Joint *J2* would **not** be able to act because it does not have the *turn* on the Link. Keeping the *go*-permission on Joint *J2*, we give Joint *J1* *go*-permission ($J1.go := TRUE$) also, making the network run.

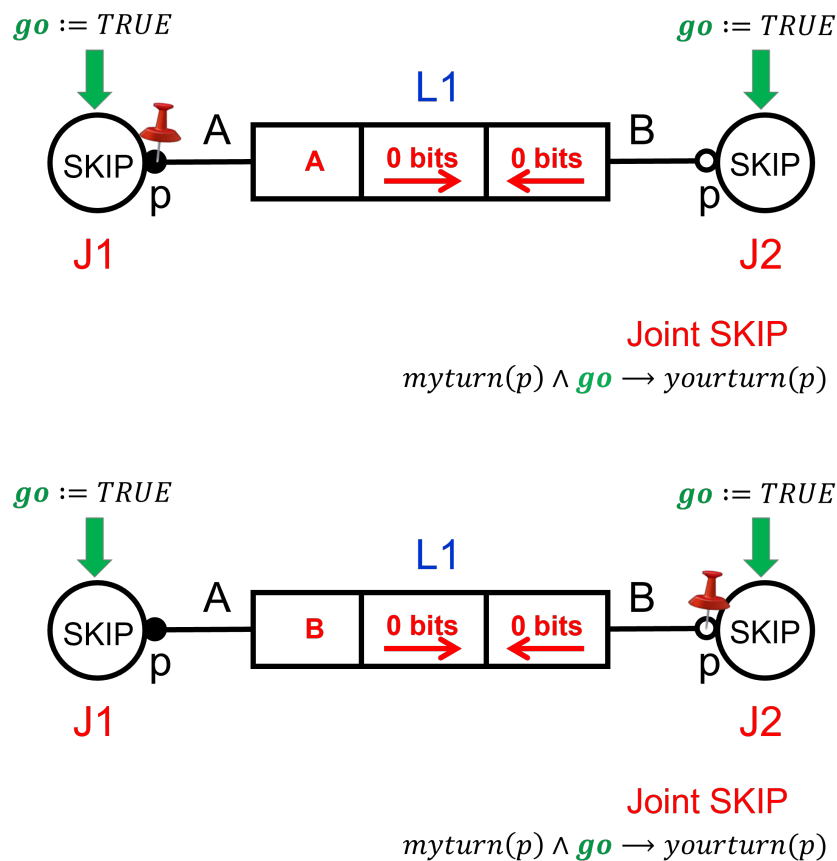


Figure 3.7: The two states that the Ping-Pong goes back and forth on. Note that the turn, marked by the red-colored pin, “goes back and forth” between ports A (top) and B (bottom).

The image at the top of Figure 3.7 shows the network at the start of its run. Because Joint $J1$ has both the *turn* and *go*-permission, it can take action, which it does by giving the *turn* to Joint $J2$ as specified in the guarded command of Joint *SKIP*. When Joint $J2$ gets the *turn* and has *go*-permission, it gives the *turn* back to Joint $J1$. So, the red pin moves back and forth between the Joints, thereby having a ping-pong motion. See Figure 3.7.

The back-and-forth passing of *turn* continues until at least one *go*-permission is removed; see Figure 3.8.

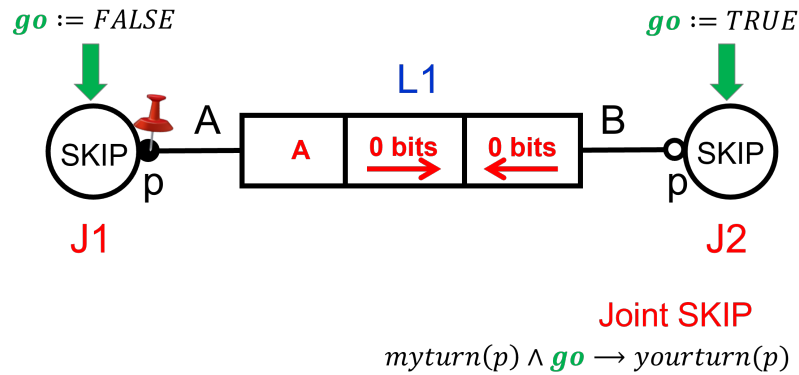


Figure 3.8: Final state of the Ping-Pong run, after we make the go signal in Joint $J1$ FALSE.

3.3 Flexible Initialization

As discussed in Section 2.1, components in an asynchronous circuit use local communication to signal readiness to communicate, output data validity, and receipt of input data. These local communications (also known as handshake signaling or simply protocols) replace the global clock that sequences (global state update) operations periodically in synchronous circuits [66]. Asynchronous protocols go beyond what clocked updates can do by supporting non-periodic local state updates distributed over the system on a when- and where-needed basis.

In a handshake, one unit is *active* (that is, it initiates the handshake by issuing a request to the other unit), and the other is *passive* (it receives the request and replies – when it is ready – with an acknowledgment). These *active* and *passive* communication protocols also apply to Link *ports*. Specifically:

1. *active* Link port :- starts the communication.
2. *passive* Link port :- responds to the communication.
3. *push* Link :- moves data from its active port to its passive port.
4. *pull* Link :- moves data from its passive port to its active port.

Active-passive and push-pull protocol settings determine whether the system is functional and how well it performs. As in prior research at Caltech [6], Philips Research [72] and the University of Manchester [2], we denote the active port with a “bullet” (\bullet) and the passive port with a “circle” (\circ). In contrast to these works, **we discovered that protocol settings do not need to be fixed** [16]. With the Link-Joint model, we can implement any given active-passive and push-pull protocol setting that a designer or compiler may assign to a system of Links and Joints *merely by initializing Link storage!*

To illustrate the flexibility in protocol settings that the Link-Joint model brings about, we use the example of a first-in-first-out buffer (FIFO). We create a FIFO in Figure 3.9 with two Links, $L1$ and $L2$, and three Joints: SRC , $COPY$ and SNK . Joint SRC generates a value and writes it into Link $L1$. Joint $COPY$ reads the value in Link $L1$ and writes it into Link $L2$. Joint SNK consumes a value that it reads from Link $L2$.

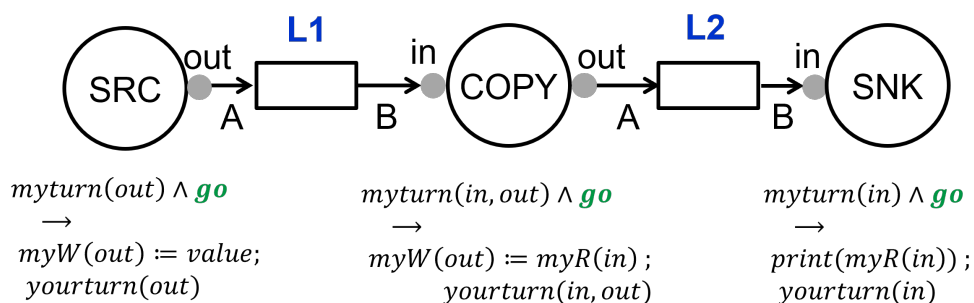


Figure 3.9: A simple FIFO with unspecified active-passive protocol.

The active-passive protocol setting does not need to be static for the FIFO to function. Each Joint does not need to have a fixed protocol setting for its ports. In Figure 3.9, we color each port grey to denote an unspecified protocol setting. The Link-Joint model combines basic parts into larger designs and ultimately into systems, independent of initial states these may assume. With this flexibility, the FIFO design

of Figure 3.9 can start in any of four different initial configurations (Figure 3.10).

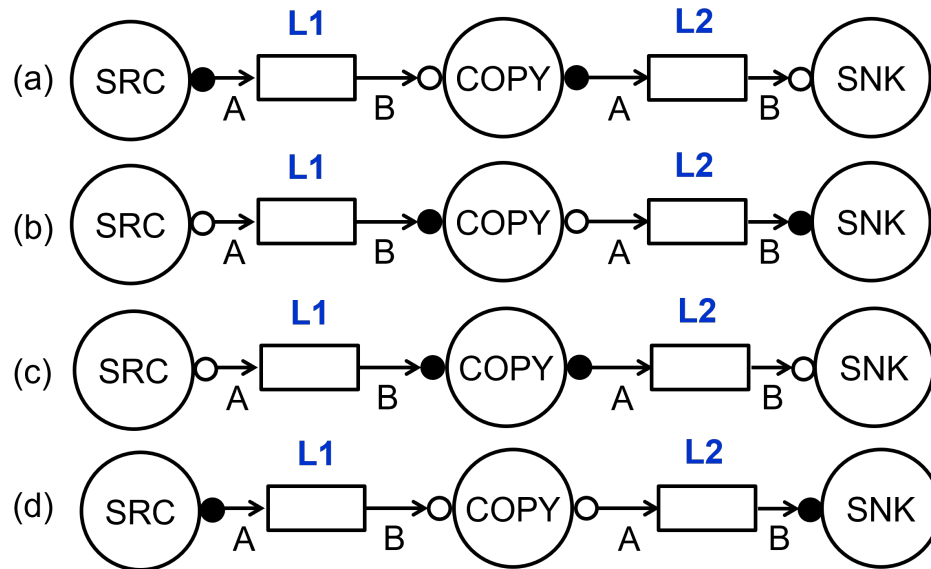


Figure 3.10: The simple FIFO in figure 3.9 with four different initial active-passive protocol setting configuration. Note that Links L1 and L2 each have one active (\bullet) and one passive (\circ) port.

In addition, there is no need for distinct *push* or *pull* Link specifications. Instead, we can achieve any desired initial protocol configuration by initializing the Links' *turn* variables. Therefore, Link ports that initially have the *turn* and start the communication are *active* ports, while Link ports initially lacking the turn are *passive* ports in the configured Link. Had we used “fixed protocol settings,” Figure 3.3 would require *two* separate module definitions to define Links, Figure 3.4 would require four individual module definitions to define Joint *COPY*, and Figure 3.9 would require *not one* but *four* different FIFO designs, each with a different active-passive setting. Thus, with flexible protocol settings, a design requires fewer models and fewer library elements, which results in fewer parts to verify, validate, simulate, compile, implement, test, and debug.

Figures 3.11 and 3.12 show a snippet of the execution (three stages in an inter-

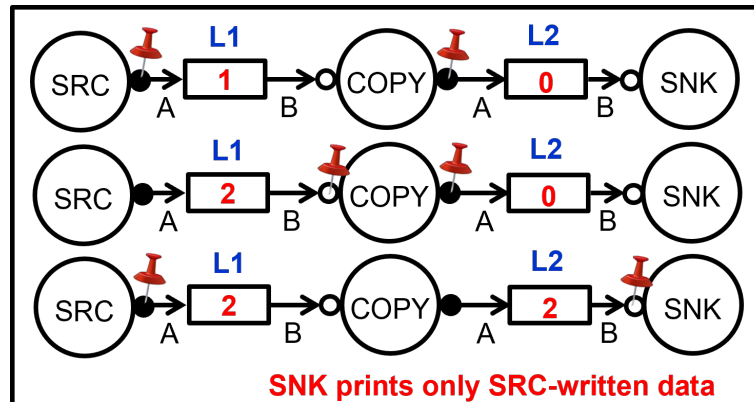


Figure 3.11: Snippet of execution from the initial active-passive configuration in Figure 3.10(a). With this initial configuration, only Joint SRC can act because it has the *turn* on Link L1, while Joint COPY and Joint SNK wait until they have all the *turns* required for them to act (top). Joint SRC acts by writing over the data in Link L1 with new data, 2, and relinquishing its turn on L1 (middle). Joint COPY (middle) now stops waiting because it has all the Link turns necessary to act. It acts by copying L1's data value, 2, to L2 and relinquishing both Link turns (bottom). Joint SNK finally has the *turn* on Link L2, so it can read and print out data value on Link L2.

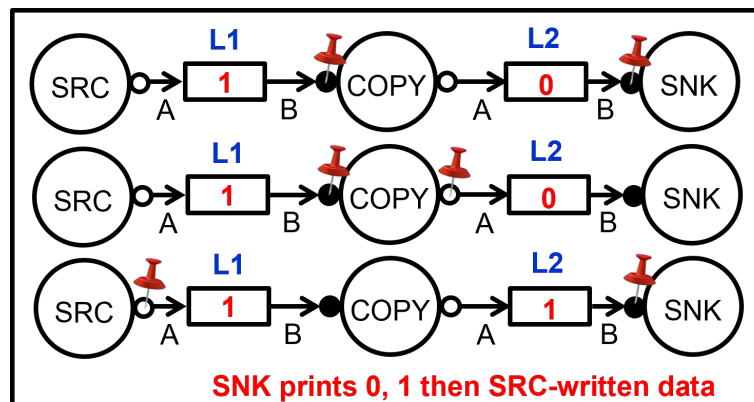


Figure 3.12: Snippet of execution from the initial active-passive configuration in Figure 3.10(b). With this initial configuration, only Joint SNK can act because it has the *turn* on Link L2, while Joint COPY and Joint SRC wait until they have all the *turns* required for them to act (top). After Joint SNK has read and printed L2's data value, 0, and relinquished its turn on L2 (middle), Joint COPY now has all the Link turns necessary for it to act. It acts by copying L1's data value, 1, to L2 and by relinquishing its turn on Link L1 and its turn on Link L2 (bottom). Joint SRC finally has the *turn* on Link L1 so it can write a new data value on Link L1 in the next step.

leaving “run”) with the first two initial configurations, Figure 3.10(a) and (b), and describe the operation of the FIFOs using red-colored pins, as introduced earlier in Figure 3.6, to track which Link port has the turn. Remember: the port with the pin has the turn. Figures 3.13 and 3.14 show the execution of the other two initial configurations.

Initialization has consequences. For this simple FIFO example, each initial active-passive configuration (Figure 3.10) produced a different output trace at Joint *SNK*. The differences in initialization may not matter so much for this FIFO but can be crucial. For instance, differences in initialization are crucial when measuring throughput versus occupancy of ring FIFOs or when characterizing the performance of a protocol, circuit, or manufacturing process. See Chapter 6 for more details on throughput analysis. Moreover, being able to initialize each Link’s *turn* and data variables gives the designer flexibility for exploring behavior for regular execution of the design and irregular behavior in test and debug instances.

Links and Joints provide a general model that facilitates design reuse. Flexible binding of settings as well as of choice of protocol and even circuit family gives Link-Joint systems enjoyable simplicity in design, design process, and even test, debug, and analysis [16].

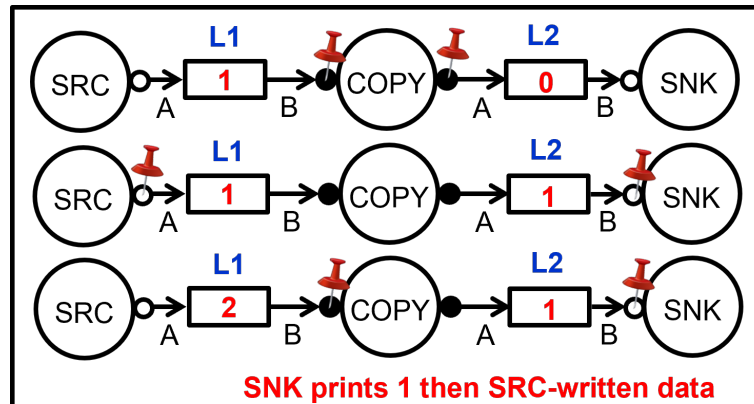


Figure 3.13: Snippet of execution from the initial active-passive configuration in Figure 3.10(c). With this initial configuration, Joint COPY has all the Link turns necessary to act. It acts by copying Link L1's data, 1, to Link L2 and relinquishing both Link turns (middle). Now, both Joint SRC and SNK can act because they each have the turn on L1 and L2, respectively (middle). Though SRC acts before SNK here, they may act in either order or at once.

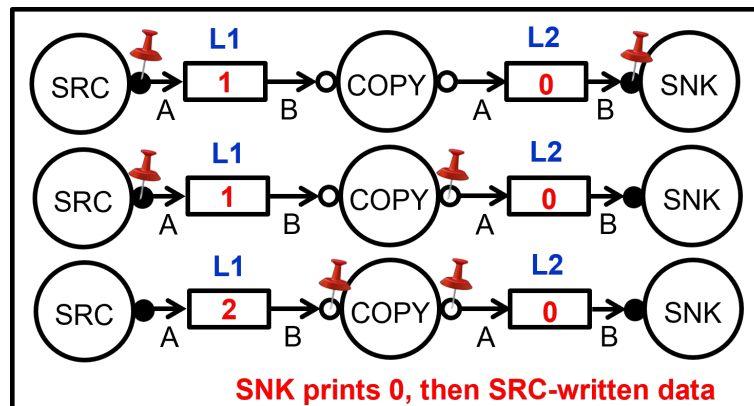


Figure 3.14: Snippet of execution from the initial active-passive configuration in Figure 3.10(d). With this initial configuration, both Joint SRC and SNK can act because they each have the turn on L1 and L2, respectively (top). Joint COPY waits until SNK has printed L2's data value, 1, and has relinquished its turn on L2 (middle). COPY also waits until SRC has written a new data value, 2, into Link L1 and relinquished its turn on L1 (bottom). Though SNK acts before SRC here, they may act in either order or at once.

3.4 Executable model in Verilog

The formal specification of the Link-Joint network with the shared variable model and guarded commands is an excellent way to reason about the network. However, to validate the design’s behavior for regular operation or test and debug, we need a representation that can be simulated. We developed Verilog behavioral modules to capture the formal semantics. We chose Verilog because it is very popular with hardware designers, and its module interface and connection reflect how we implement Links and Joints at the circuit level [59]. Therefore, Verilog continues to suit our simulation and validation needs as we make the Link-Joint network less abstract by refining it with more protocol, data encoding, or circuit implementation details; see Chapter 5.

In Verilog, modules are typically self-contained descriptions that encapsulate certain functionalities. Modules encapsulate design hierarchy and communicate with other modules through a set of declared input, output, and bidirectional *pins*¹. **Our shared variable model appears differently in Verilog because Verilog uses a communication interface rather than shared variables.** Link variables are now local to the Link module and can be changed by the Link based on requests from its two Link ports.

Figure 3.15 gives a Verilog behavioral module of a Link. It has multiple pins to emulate the connection to the Link variables (line 2 of Figure 3.15). It has pins *Amyturn*, *Ayourturn*, *Bmyturn*, and *Byourturn* to connect to Link variable *turn*. Wires at pins *Amyturn* and *Bmyturn* carry signals from the Link to the Joints (or environment) at ports *A* and *B*, respectively, to indicate whose turn it is on the Link. Only one of these signals is high (TRUE or logic 1) at a time.

¹*Pins* are originally called *ports* in Verilog, but in this dissertation, we call them *pins* to avoid confusion with *ports* in the Link-Joint model.

```

1  /*          Link Description          */
2  module Link (Amyturn, Ayourturn, Bmyturn, Byourturn, ABin,
   About, BAin, BAout, init_turn, init_AB, init_BA);
3  parameter bw = 32;
4  input      Ayourturn, Byourturn, init_turn;
5  input      [bw-1:0] ABin, BAin, init_AB, init_BA;
6  output     Amyturn, Bmyturn;
7  output     [bw-1:0] ABout, BAout;
8  reg       reg_Amyturn, reg_Bmyturn;
9  reg       [bw-1:0] reg_ABout, reg_BAout;
10
11  assign Amyturn = reg_Amyturn;
12  assign Bmyturn = reg_Bmyturn;
13  assign ABout  = reg_ABout;
14  assign BAout  = reg_BAout;
15
16  initial begin
17      reg_Amyturn = !init_turn;
18      reg_Bmyturn =  init_turn;
19      reg_ABout  =  init_AB;
20      reg_BAout  =  init_BA;
21  end
22
23  always @(posedge Ayourturn) begin
24      #10 reg_ABout  = ABin;
25          reg_Amyturn = 1'b0;
26          reg_Bmyturn = 1'b1;
27  end
28
29  always @(posedge Byourturn) begin
30      #10 reg_BAout  = BAin;
31          reg_Bmyturn = 1'b0;
32          reg_Amyturn = 1'b1;
33  end
34  endmodule

```

Figure 3.15: Verilog behavioral module for a Link. Note that Verilog is a timed model, therefore we inject time-ticks (`#10`) to ensure all the changes have been made before the next action. This implementation is used only for simulation and validation. In Chapter 7, circuit implementations have their cell library and gate-level timing.

Wires at pins *Ayourturn* and *Byourturn* carry signals from the Joints (or environment) at ports *A* and *B*, respectively, to the Link to inform the Link that it is done with the *turn* and the *turn* can be given to the Link's other port. Wires at pins *ABin*, *ABout*, *BAin*, and *BAout* carry data in and out of the Link. Pin *init_turn* is used to set which port has the *turn* initially, while pins *init_AB* and *init_BA* are used to give initial values to the data stored in the Link.

The Verilog registers *reg_Amyturn* and *reg_Bmyturn*, declared on line 8 of Figure 3.15, store the internal state of the Link's *turn* variable. As the network executes, these registers drive the wires at pins *Amyturn* and *Bmyturn*.

The registers *reg_ABout* and *reg_BAout*, declared on line 9 of Figure 3.15, store data variables, *data_{AtoB}* and *data_{BtoA}* respectively. When the Joint at Link port *A* writes data, (*myW(A)*), the data come into the Link module over pin *ABin* on line 2 and get stored in *reg_ABout* before the *turn* is given back (lines 24-26 of Figure 3.15). Likewise, writes from Link port *B*, (*myW(B)*), come into the Link module over the *BAin* pin and get stored in *reg_BAout* before the *turn* is given back (lines 29-31 of Figure 3.15).

Read requests from Link port *A*, (*myR(A)*), get their data from the Link module over pin *BAout*, which is driven by *reg_BAout* on line 14 of Figure 3.15. Likewise, read requests from Link port *B*, (*myR(B)*), get their data from the Link module over pin *ABout*, which is driven by *reg_ABout* on line 13 of Figure 3.15.

Figure 3.16 gives a Verilog behavioral module of a Joint *COPY*. Its pins model the interface for interaction with its connected Links. When high, the input wire at pin *in_myturn* notifies the Joint that it has the *turn* on the Link connected at port *in*; likewise for the input wire at pin *out_myturn*. The Joint uses the output wires at pins *in_yourturn* and *out_yourturn* to notify the Links at ports *in* and *out* that it is relinquishing each *turn*. Pin *in_myR* gets the data coming from the Link connected

at port *in* to the Joint, and *out_myR* pin carries out the data leaving the Joint to the Link connected at port *out*.

The guarded command of Joint *COPY* applies directly. Its guard in the shared variable model (Figure 3.4) is the **if condition** checked on line 19 of Figure 3.16 Verilog module. After the copy action, $reg_out_myW = in_myR$ on line 20, the Joint indicates that it relinquishes the *turn* to the Links at ports *in* and *out* as accomplished on lines 21-24 of Figure 3.16. Recall that the execution of the commands must appear

```

1  module COPY (in_myturn, in_yourturn, in_myR, out_myturn,
   out_yourturn, out_myW, go);
2  parameter bw = 32;
3  input      in_myturn, out_myturn, go;
4  input      [bw-1:0] in_myR;
5  output     in_yourturn, out_yourturn;
6  output     [bw-1:0] out_myW;
7  reg       reg_in_yourturn, reg_out_yourturn;
8  reg       [bw-1:0] reg_out_myW;
9
10 assign in_yourturn  = reg_in_yourturn;
11 assign out_yourturn = reg_out_yourturn;
12 assign out_myW      = reg_out_myW;
13
14 initial begin
15     reg_in_yourturn  = 1'b0;
16     reg_out_yourturn = 1'b0;
17 end
18 always @(in_myturn or out_myturn or go) begin
19     if (in_myturn && out_myturn && go) begin
20         #10 reg_out_myW      = in_myR;
21         reg_in_yourturn     = 1'b1;
22         reg_out_yourturn    = 1'b1;
23         #10 reg_in_yourturn  = 1'b0;
24         reg_out_yourturn    = 1'b0;
25     end
26 end
27 endmodule

```

Figure 3.16: Verilog behavioral module for Joint *COPY*.

as a single atomic action. Hence, lines 21-24 of Figure 3.16 must run as one atomic action. This atomicity is achieved by the matching #10 delays on lines 24 and 30 in Figure 3.15 and line 23 in Figure 3.16. Note that the registers in the Joint module are Verilog artifacts and have nothing to do with the state.

We have created a library of abstract behavioral modules in Verilog for a Link and various Joints. Using this library, we can create a behavioral module for Link-Joint networks by instantiating the modules in the library. Figure 3.17 gives the instantiation and simulation sequence for the simple FIFO in Figure 3.9.

The Verilog testbench in Figure 3.17 shows the connections between the Link instances and instances of Joints *COPY*, *SRC*, and *SNK*. Figure 3.18 shows the waveforms from running the testbench in Figure 3.17.

3.5 Chapter Contributions

While the Link-Joint paradigm predates this dissertation and the research effort is joint work with my supervisor, the following key contributions are largely mine.

- I extended the original partially defined semantics of Links and Joints to a fully defined (though not yet fully formal) model based on shared Link variables and Joint guarded commands. This extension includes the notion of ports as physical Link-Joint connections and guarded command terms.
- I identified that initialization determines protocol settings such as active-passive and push-pull in Link-Joint networks, in contrast to previous knowledge. Flexible initialization rather than static circuitry determines who communicates first.
- I developed and implemented a Link and Joint abstract behavioral module library in Verilog.

```

1  module FIFO;
2      reg    [2:0] go;
3      // declare wire connections with more than 1 bit
4      wire  [7:0] L1_ABin, L2_ABin, L1_About, L2_About;
5
6      initial begin    //simulation sequence
7          $dumpvars();
8              go [2:0] = 3'b000;
9          #10 go [2:0] = 3'b111;
10         #60
11         $finish();
12     end
13
14     SRC #(.init_value(1)) J1
15     (.myturn(L1_Ame), .yourturn(L1_Ayou), .myW(L1_ABin), .
16     go(go[0]));
17     Link L1
18     (.Amyturn(L1_Ame), .Ayourturn(L1_Ayou), .Bmyturn(
19     L1_Bme), .Byourturn(L1_Byou), .ABin(L1_ABin), .ABout(
20     L1_About), .BAin(X), .BAout(X), .init_turn(1'b0), .
21     init_AB(1), .init_BA(X));
22     COPY J2
23     (.in_myturn(L1_Bme), .in_yourturn(L1_Byou), .in_myR(
24     L1_About), .out_myturn(L2_Ame), .out_yourturn(L2_Ayou),
25     .out_myW(L2_ABin), .go(go[1]));
26     Link L2
27     (.Amyturn(L2_Ame), .Ayourturn(L2_Ayou), .Bmyturn(
28     L2_Bme), .Byourturn(L2_Byou), .ABin(L2_ABin), .ABout(
29     L2_About), .BAin(X), .BAout(X), .init_turn(1'b0), .
30     init_AB(0), .init_BA(X));
31     SNK J3
32     (.myturn(L2_Bme), .yourturn(L2_Byou), .myR(L2_About),
33     .go(go[2]));
34 endmodule

```

Figure 3.17: A Verilog testbench for a simple FIFO (Figure 3.9). We have initialized the active-passive setting in this testbench to match Figure 3.10(a), such that the execution of Link *L1* starts at Link port *A* ($L1.init_turn(1'b0)$) and Link *L2* starts at Link port *A* ($L2.init_turn(1'b0)$). We can start with any active-passive initial setting by giving a value to pin *init_turn* of each Link.

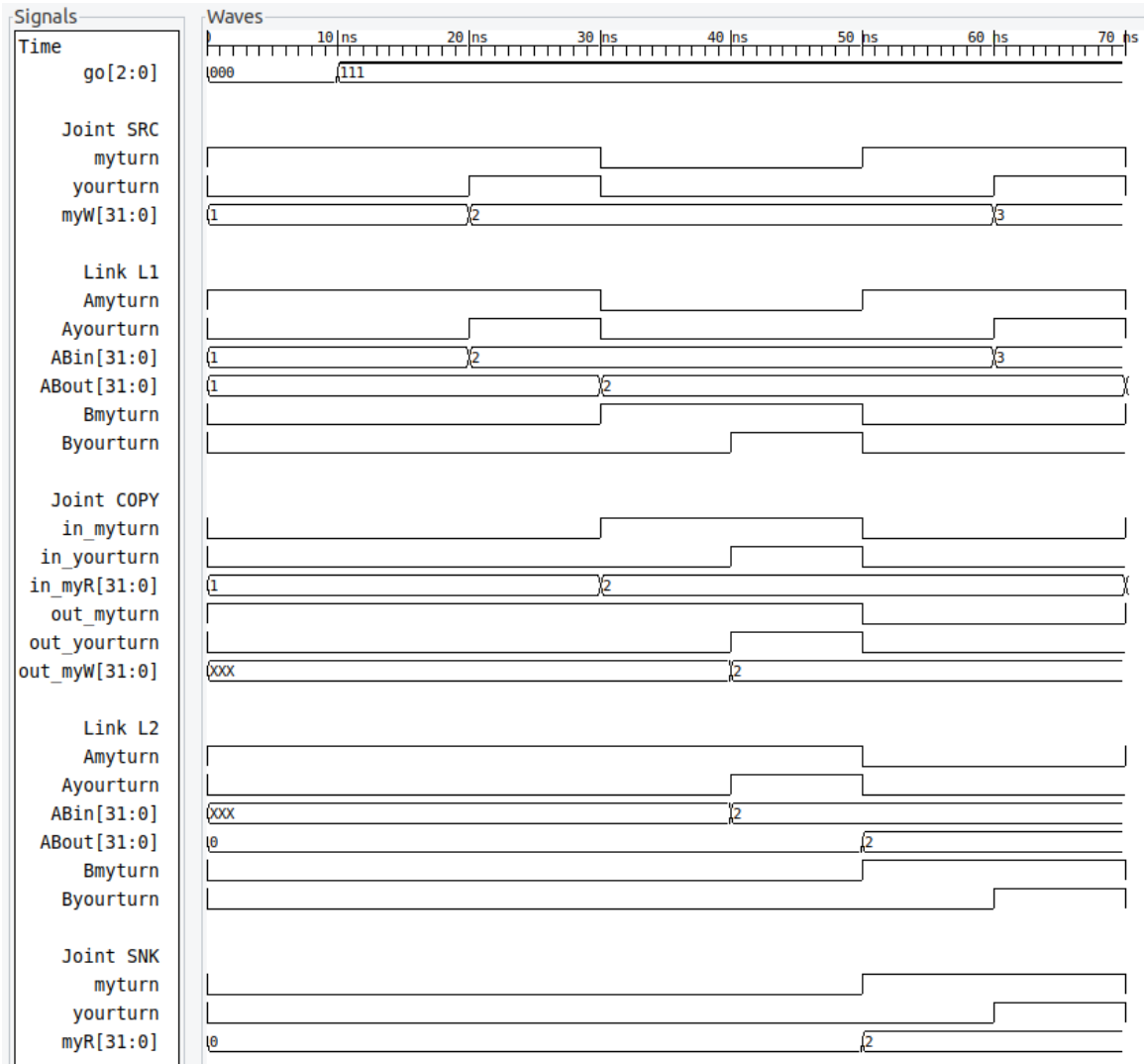


Figure 3.18: Waveforms for simulating Verilog testbench in Figure 3.17. The waveforms reflect the snippet of execution shown in Figure 3.11, showing that Joint SNK consumed data values generated by Joint SRC while the initial values 1 (in Link L1) and 0 (in Link L2) are overwritten.

Chapter 4: Compilation

The Link-Joint specification is *sufficiently abstract* to accommodate a variety of circuit implementations with different protocols, signaling logic, data representations, circuit families, and fabrication means. Before this dissertation, Link-Joint designs were hand-constructed. Links and Joints lacked an easy-to-use front-end design entry point, which has limited their usage and user base. There was also no tractable way to design and test large systems.

To address these issues, this chapter details a front-end solution for Links and Joints, which we presented and published in a paper [17] at the 2023 ASYNC conference. Our solution follows the same strategy as both past [2–6, 49, 50, 72] and present [1, 32, 33] works as discussed in Section 2.3, which use **Syntax-directed Translation** to compile designs specified in a high-level programming language into a network of *handshake circuits* connected by *channels*. The handshake protocol and circuit family are determined before compilation in each referenced work.

Our solution uses syntax-directed translation to compile designs spelled out as programs (ACT programs) into Link-Joint networks; see Figure 4.1. Our method differs in that we *bind implementation decisions as late as possible* because our targets are circuit-neutral Link-Joint networks. Because we compile programs **not** into networks of channels and handshake circuits but into networks of Links and Joints, our key challenge – and a significant part of the work – was developing a new target representative of handshake circuits in terms of Links and Joints.

This chapter introduces the languages in our source programs and the target Link-Joint library elements. We demonstrate our compilation with examples.

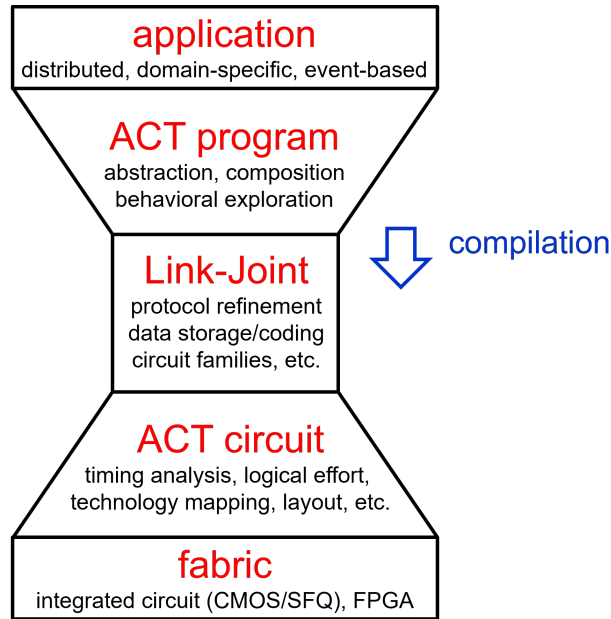


Figure 4.1: Compilation as a focus in our design flow, *Onà*.

4.1 Source Programs

Transforming the design process into a programming activity makes designing with Links and Joints more accessible, even though Links and Joints are already an abstraction. We express our designs in languages that support message passing for synchronization and communication. As mentioned in Section 1.1.1, our front-end solution branches off Yale’s Asynchronous Circuit Toolkit (ACT) — an electronic design automation framework [1,32,33]. To get to Link-Joint networks, we compile ACT programs with data-flow parts written in ACT sublanguage *dataflow* and control-flow parts in CHP (*Communicating Hardware Processes*) [32].

4.1.1 The Communicating Hardware Processes (CHP) Sublanguage

CHP [42] is based on a combination of Hoare’s *Communicating Sequential Processes* (CSP) [23] and Dijkstra’s guarded command language [15]. CHP supports state-

ments, communication constructs, composition operators, conditional execution, and iteration. The documentation [34] gives the full details on the CHP language. Here is a summary of the CHP statements used in this dissertation and their notation:

| | | |
|--------|--|---|
| $S :=$ | $skip$ | no-op |
| | $ v := E$ | assign E 's result to v |
| | $ A ! E$ | send E 's result via A |
| | $ A ? v$ | receive value for v via A |
| | $ \#A$ | probe channel A |
| | $ S_1 ; S_2$ | sequential composition |
| | $ S_1 , S_2$ | internal parallel composition |
| | $ [G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n]$ | deterministic select |
| | $ [G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n []$ $else \rightarrow S_{n+1}]$ | deterministic select with explicit <i>else</i> case |
| | $ [[G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n]]$ | nondeterministic select |
| | $ [[G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n []$ $else \rightarrow S_{n+1}]]$ | nondeterministic select with explicit <i>else</i> case |
| | $ *[G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n]$ | deterministic loop |

We use v to denote a variable and A to denote a channel. We use S, S_1, S_2, S_n to denote statements, E to denote expressions, and G_1, G_n to denote guards. The expression syntax in CHP is similar to expressions in the C language. The expressions include numerical or Boolean expressions using arithmetic, comparison, Boolean, logical and bit operators, and function calls on expressions. Guards are expressions with a Boolean result. For communication and synchronization, CHP has *send(!)*, *receive(?)*, and *probe(#)* constructs over channels. A *probe* is an explicit Boolean signal that reports if the communication partner at the other end of the point-to-point channel is ready to communicate. Unlike a probe, which senses but remains uncommitted, *send*

and *receive* wait until a communication partner “shows up.” CHP expressions may use probes. Currently, probes are permitted only in guard expressions of selection statements. CHP statements are executed in sequence or parallel.

When a guard is TRUE in deterministic and nondeterministic select statements, the corresponding statement may be executed. A selection is deterministic because the program uses the deterministic select syntax to assert that, at most, one guard can become TRUE per selection [34]. In a nondeterministic select statement, multiple guards may become TRUE. If no guard is TRUE in the select statement execution, the program waits until a guard becomes TRUE before execution continues. To avoid being stuck, one can add an **else** case to the statement. For example, a deterministic guarded command of the form $[G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n [] \textit{else} \rightarrow S_{n+1}]$ executes S_{n+1} if all guards are FALSE. Unlike CSP, where guards may have *send* or *receive* constructs but not *probes*, CHP guards in select statements may have *probes*, but not *sends* or *receives*. Loops can have multiple guards; however, the program exits the loop when none of its guards is TRUE. In CHP loop statements, *probes* can NOT be used in the guards; *probes* are not guaranteed to be stable. In particular, a probe that evaluates to TRUE remains TRUE, but a probe that is FALSE can become TRUE during guard evaluation.

4.1.2 The Dataflow Sublanguage

Dataflow is a data-oriented language for creating pipelined or streaming asynchronous circuits. It exclusively uses expressions with channels as the basic terms for defining the dataflow computations and network connections [36, 70]. A dataflow program has the form

$$\textit{dataflow} \{S_1; \dots; S_n\}$$

where S_1 and S_n are dataflow statements and “;” is just a statement separator. These statements run in parallel. Dataflow statements are mostly in the following forms:

- **Function Statement:** This statement computes a given function on data from input channels and passes the results to an output channel. For example, $dataflow \{A * B \rightarrow C\}$ repeatedly reads and multiplies each data item from channels A and B and writes the result to channel C .
- **Conditional Split Statement:** This statement sends data from one input channel to one of several output channels based on a condition. For example, $dataflow \{ \{c\} I \rightarrow O_0, O_1, \dots, O_n \}$, sends data from input channel I to output channel O_j if the condition c evaluates to the index value j .
- **Conditional Merge Statement:** This statement routes data from one of several input channels to one output channel based on a condition. For example, $dataflow \{ \{c\} I_0, I_1, \dots, I_k \rightarrow O \}$, sends data from channel I_j to channel O if the condition c evaluates to index value j .

Recall that Joints compute while Links store, but with *Dataflow*, the primitives (statements) both compute and store while the channels are solely for transporting data. Therefore, channels differ from Links. The Link-Joint model is versatile enough to represent both control- and data-oriented programs.

4.2 Link-Joint Library Elements

Our compilation process starts with source programs with either CHP or dataflow parts and ends in Link-Joint networks. As mentioned in the introduction of this chapter, one of the key challenges with our compilation is developing target Link-Joint elements.

A Link has minimal variation in its specification, only changes in its data widths and initial (active versus passive) port settings. In contrast, there are multiple types of Joints based on their functionality and flow control. We have broadly classified the types of Joints based on the part they commonly play in the network. These include:

1. **Computation Class**, which includes Joints involved in computation in control-oriented programs such as variables (Joint VAR), expressions (Joint E), and data transfers such as assignments (Joint TRF).
2. **Shared Resources Class**, which includes Joints for sharing structures, multiplexers (Joints RMUX, WMUX, MUX), and Joints for sequencing Joint execution commands (Joint FSM).
3. **Flow Control Class**, which includes Joints that manage the flow of control in the compiled Link-Joint network, such as sequencers (Joint SEQ), parallel compositors (Joint PAR), loops (Joint REP), and selections (Joint SEL).
4. **Communication Class**, which includes Joints for synchronization and communication such as channels (Joint CHAN) and probe expression evaluations (Joints $E_{waitcycle}$, $Wait_{cycle}$)
5. **Dataflow Class**, which includes Joints for dataflow or streaming operations such as Joints COPY, SRC, and SNK.

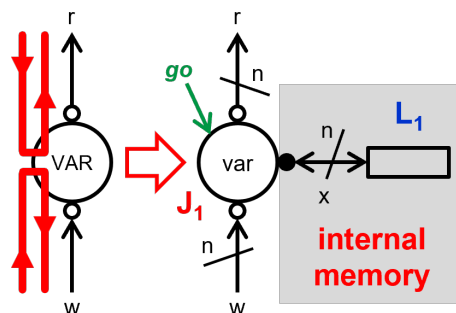
The following subsections give further details on the classified Joints, showing their representation and guarded command specifications. We mark the flow control for each Joint with fat line-arrows colored in red, blue, and purple to separate paths in the Joint figures. Also, note that the ports for each Joint are colored black or white to indicate the default initialization of Link states for normal operation in the compiled ACT-CHP program context. It is important to be reminded that *this*

default initialization of the active-passive or push-pull setting is **not** frozen. The flexible initialization property, discussed in Section 3.3 of the Link-Joint model, still remains valid.

4.2.1 Computation Class - Joints *VAR*, *E*, *TRF*

This class focuses on Joints involved in data operations in control-oriented programs.

Joint VAR: *VAR* in Figure 4.2 represents an n -bit program variable, $n \geq 0$. *VAR* has a basic Joint *var* with read and write ports r , w for mutual exclusive use. Because there is no storage in Joints, we store the variable's value in Link L_1 connected to internal *VAR* port x . *VAR* has two guarded commands, one for read and one for write. Neither relinquishes the turn on x , which we can set externally in L_1 for initialization and test purposes.



$$\begin{aligned} myturn(r, x) \wedge go &\rightarrow myW(r) := myR(x) ; yourturn(r) \\ myturn(w, x) \wedge go &\rightarrow myW(x) := myR(w) ; yourturn(w) \end{aligned}$$

Figure 4.2: Joint VAR library element. Note that VAR is the representation in a Link-Joint network, while var is the basic Joint inside VAR showing all the internal connections. The red fat line-arrows mark the flow of control in Joint VAR.

Joint E: Joint $E(\text{expression})$ evaluates expressions in communication statements, assignments, and probe-less guards of guarded command statements. It has startup and output port c , ports e_1 to e_m for connecting expression variables v_1 to v_m (see Figure 4.3). Like the data transfer Joint TRF in the next paragraph, an internal

FSM – connected at port x of Joint E – is used to sequence two commands. The evaluation of the expression uses $f(myR(e_1..e_m))$ where f is the computation and $myR(e_i)$ substitutes v_i in f . Like in Figure 4.2, the fat line-arrows mark the flow of control in Joint E . We use red and blue to separate overlapping line-arrows.

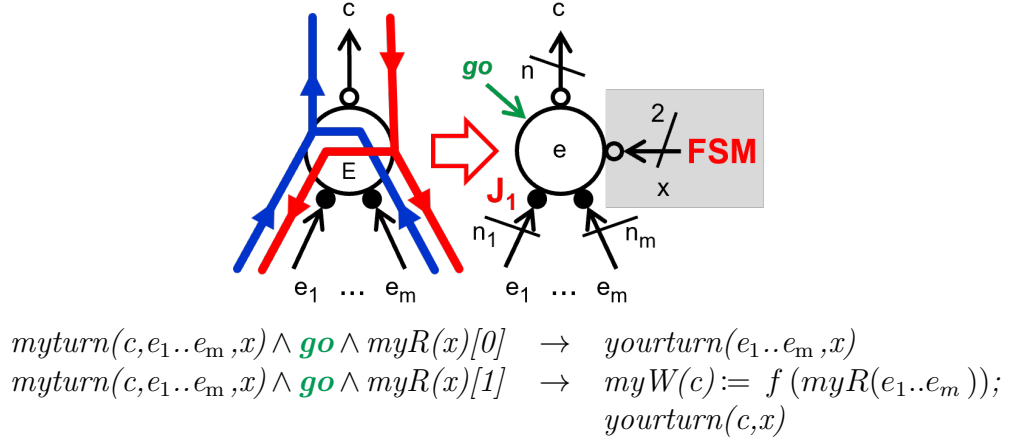
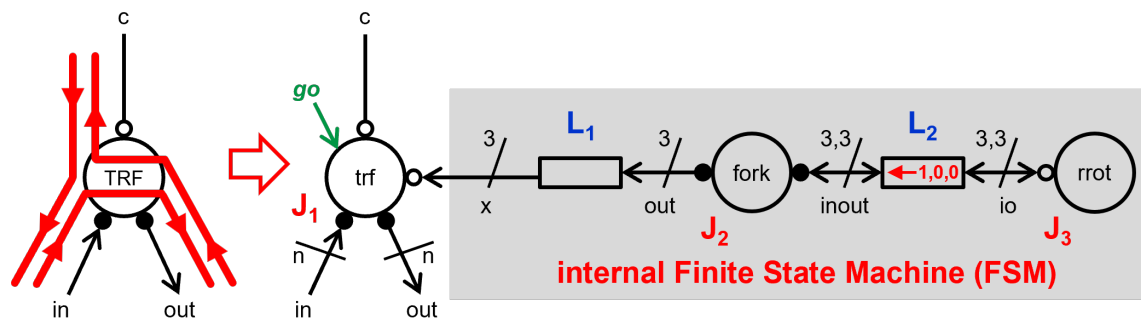


Figure 4.3: Joint E library element.

Joint TRF: This is the data transfer Joint. Data transfer is not restricted to CHP assignments ($v := E$) alone. Also, data communications over CHP channels are data transfers. A *receive* communication ($A?v$) saves the data transferred over the channel into a variable, and a *send* communication ($A!E$) transfers the result of a computed expression over the channel. As a result, the *TRF* in Figure 4.4 is generated as part of both a CHP assignment and a CHP communication statement.

TRF has a basic Joint *trf*, and ports c , in , out . When prompted by c , *TRF* first requests n bits of data, $n \geq 0$, from variables or channels connected to port in , which it then “transfers” to variables or channels connected to port out , before reporting completion at c . This sequence is controlled by 1-hot 3-bit string $myR(x)$, generated by a Finite State Machine (FSM) connected to internal *TRF* port x and discussed next in Section 4.2.2.



$$\begin{aligned}
 & \text{myturn}(c, in, out, x) \wedge \text{go} \rightarrow \\
 & \quad \text{myR}(x)[0] \rightarrow \text{yourturn}(in, x) \\
 & \quad \text{myR}(x)[1] \rightarrow \text{myW}(out) := \text{myR}(in) ; \text{yourturn}(out, x) \\
 & \quad \text{myR}(x)[2] \rightarrow \text{yourturn}(c, x)
 \end{aligned}$$

Figure 4.4: Joint TRF library element.

4.2.2 Shared Resources Class - Joints *RMUX*, *WMUX*, *MUX*, *FSM*

This class of Joints includes Joints for sharing structures such as variables and channels and Joints for sequencing Joint execution commands.

Finite State Machine (FSM): FSM sequences actions in other Joints. The FSM in Figure 4.4 shares and stores flow control information for Joint TRF, which it maintains as a 1-hot bit string. Joint *TRF*, specified in Section 4.2.1, uses the 1-hot bit position to decide which command to execute. With three guarded commands, *TRF* requires a 3-bit FSM string. After each *TRF* execution, the FSM right-rotates the bits by one position around the string. Likewise, for any Joint that uses the FSM, its number of guarded commands determines the bit width of the FSM string.

To simplify the connection at port x of the Joint that uses the FSM, the FSM has two basic Joints, *fork* and *rrrot*, a unidirectional Link L_1 from *fork* port *out* to *TRF* port x , and a bidirectional Link L_2 between *fork* port *inout* and *rrrot* port *io*. We can initialize the FSM so that (1) the leftmost bit of the string stored in L_2 in the direction from *rrrot* to *fork* is 1-hot, and (2) *fork* has the turn on both its ports

and is ready to execute its command.

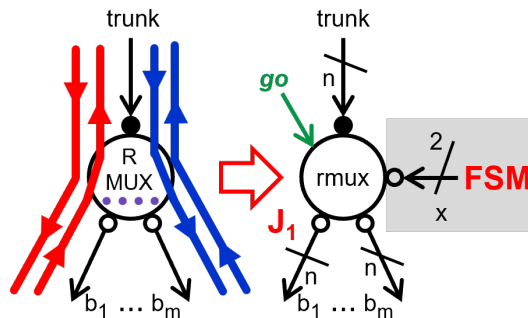
Joint *fork* copies the FSM string from L_2 to L_1 and L_2 . Its executions alternate with those of *TRF* and *rrot*. Joint *rrot* right-rotates the FSM string and returns the result to *fork*.

(**fork**): $myturn(inout, out) \wedge go \rightarrow$

$myW(inout, out) := myR(inout) ; yourturn(inout, out)$

(**rrot**): $myturn(io) \wedge go \rightarrow myW(io) := rrot(myR(io)) ; yourturn(io)$

Joints RMUX, WMUX, MUX: Multiplexing Joints provide access to a shared resource, e.g., a variable or channel, from different locations in the program. *RMUX* provides read access and *WMUX* provides write access, while *MUX* provides both. A multiplexing Joint — for read access (Figure 4.5), write access (Figure 4.6), or both (Figure 4.7) — has branch ports b_1 to b_m , one per access location, a shared access port *trunk*, and internal FSM port x to sequence its two guarded commands, using index i , $1 \leq i \leq m$. Note that a mutual exclusive constraint exists on the $myturn(b_i)$ s. Also note that *RMUX* and *WMUX* are special cases of *MUX* with zero-width data in the write and read direction, respectively.

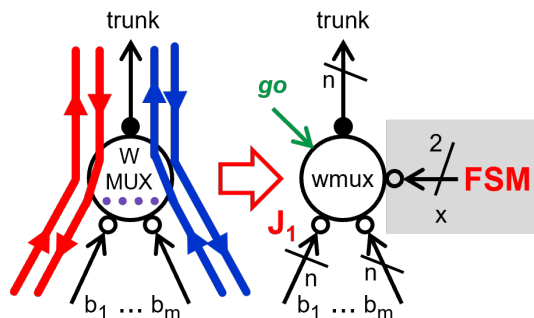


$myturn(b_i, trunk, x) \wedge go \rightarrow$

$myR(x)[0] \rightarrow yourturn(trunk, x)$

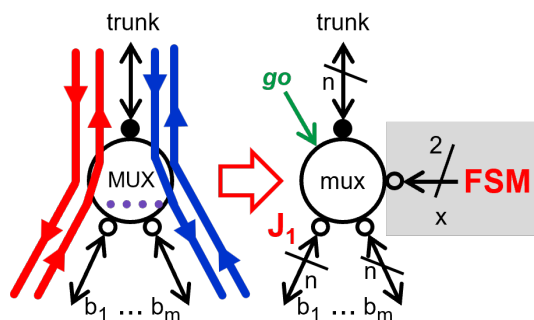
$myR(x)[1] \rightarrow myW(b_i) := myR(trunk) ; yourturn(b_i, x)$

Figure 4.5: Joint RMUX library element.



$$\begin{aligned}
 & \text{myturn}(b_i, \text{trunk}, x) \wedge \text{go} \rightarrow \\
 & \text{myR}(x)[0] \rightarrow \text{myW}(\text{trunk}) := \text{myR}(b_i) ; \text{yourturn}(\text{trunk}, x) \\
 & \text{myR}(x)[1] \rightarrow \text{yourturn}(b_i, x)
 \end{aligned}$$

Figure 4.6: Joint WMUX library element.



$$\begin{aligned}
 & \text{myturn}(b_i, \text{trunk}, x) \wedge \text{go} \rightarrow \\
 & \text{myR}(x)[0] \rightarrow \text{myW}(\text{trunk}) := \text{myR}(b_i) ; \text{yourturn}(\text{trunk}, x) \\
 & \text{myR}(x)[1] \rightarrow \text{myW}(b_i) := \text{myR}(\text{trunk}) ; \text{yourturn}(b_i, x)
 \end{aligned}$$

Figure 4.7: Joint MUX library element.

4.2.3 Flow Control Class - Joints *SEQ*, *PAR*, *REP*, *SEL*

This class of Joints manages the flow of control in the compiled Link-Joint network. These Joints generate events that trigger activity in other Joints.

Joint SEQ: *SEQ* in Figure 4.8 represents sequential composition of statements. It has basic Joint *seq*, startup port c , and ports s_1 to s_m for the m , $m \geq 1$, program statements it sequences when prompted by c . It uses an $m+1$ -bit internal FSM port x — see Section 4.2.2 — to sequence its commands, using index i , $0 \leq i < m$.

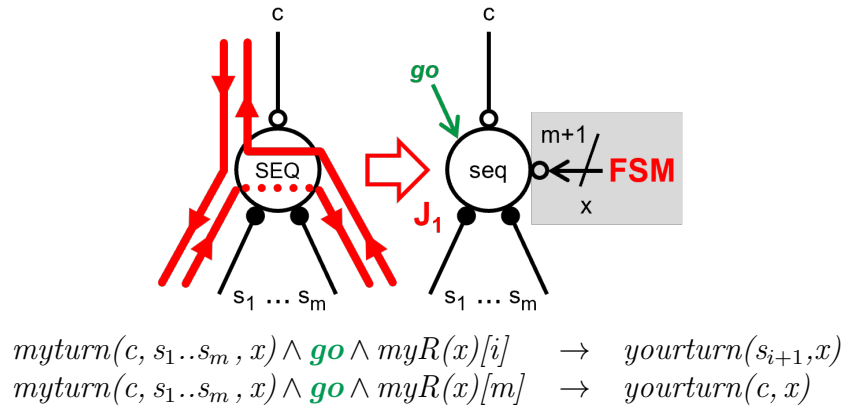


Figure 4.8: Joint SEQ library element.

Joint PAR: *PAR* in Figure 4.9 represents parallel composition. It has a basic Joint *par*, startup port c , ports s_1 to s_m for program statements it executes in parallel, and internal FSM port x to sequence its operations. *PAR* has two guarded commands.

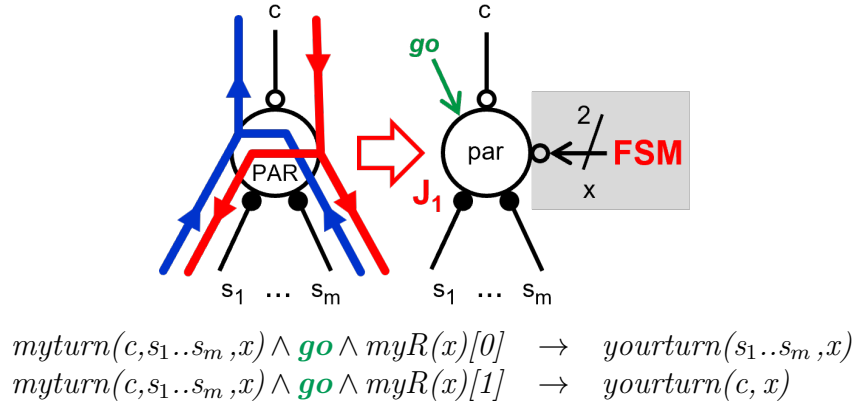
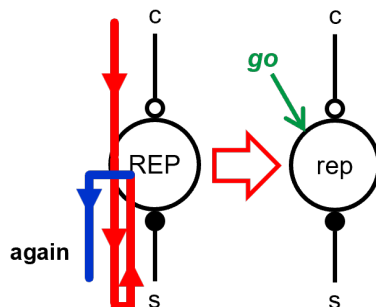


Figure 4.9: Joint PAR library element.

Joint REP: *REP* in Figure 4.10 represents infinite repetition. It has startup port c and port s for the statement it repeats. *REP* has one guarded command that never relinquishes the turn on c .

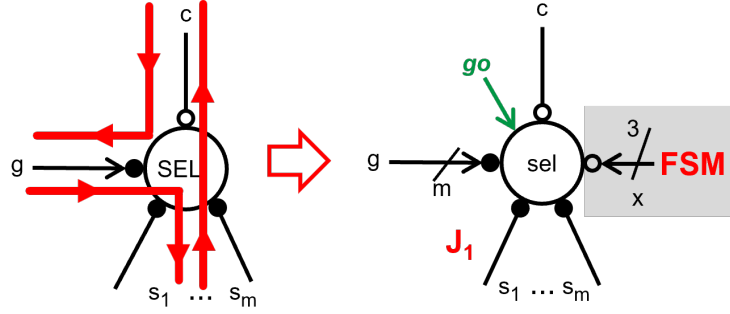


$$\text{myturn}(c, s) \wedge \text{go} \rightarrow \text{yourturn}(s)$$

Figure 4.10: Joint REP library element.

Joints SEL_{det} and SEL_{nondet} : SEL_{det} and SEL_{nondet} perform deterministic and nondeterministic selection, respectively, based on a collection of all the guards in the select statement. The guards are represented as a bit list of Booleans arriving at port g . In deterministic selection, only one guard can be TRUE at execution, but more than one guard can be TRUE in nondeterministic selection. The implementation of Joint SEL_{nondet} determines which of the TRUE guards' statements are executed. This decision can be based on round-robin, arbitration, or other selection methods. We explore guard selection implementation as a refinement in Chapter 5. Because the nondeterministic selection of guarded commands is built into the guarded command semantics [15], both deterministic and nondeterministic selections have the same guarded commands.

Our library element adheres to the CHP specification for *SELECT* without an *else* case — if **none** of the guards evaluate to TRUE, the program deadlocks, preventing any further progress. SEL_{det} and SEL_{nondet} each have basic Joints sel_{det} and sel_{nondet} (presented in Figure 4.11 as *sel*), with startup port c , ports s_1 to s_m for the program statements it may select, and internal FSM port x to sequence its operations.



$$\begin{aligned}
 \text{myturn}(c, g, s_1 \dots s_m, x) \wedge \text{go} &\rightarrow \\
 \text{myR}(x)[0] &\rightarrow \text{yourturn}(g, x) \\
 \text{myR}(x)[1] \wedge \text{myR}(g)[i] &\rightarrow \text{yourturn}(s_{i+1}, x) \\
 \text{myR}(x)[2] &\rightarrow \text{yourturn}(c, x)
 \end{aligned}$$

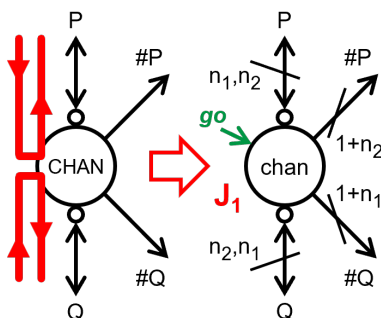
Figure 4.11: Joint SEL library element representing SEL_{det} and SEL_{nondet} .

4.2.4 Communication Class - Joints $CHAN$, $E_{waitcycle}$, $WAIT_{cycle}$

This class features Joints for synchronization and communication. Joint $CHAN$ represents a CHP channel. The other Joints in this class facilitate using *probes* in expressions. Probes are permitted only in guard expressions of selection statements.

Joint $CHAN$: Because Joints are storage-free, we represent CHP channels as Joints — *not* as Links. This makes perfect sense because channels synchronize parallel processes, and Joints excel at bringing together multiple participants by synchronizing them. Joint $CHAN$ in Figure 4.12 represents a channel with two ports, P and Q , for connecting two communicating processes that exchange n_1 data bits from P to Q , n_2 data bits from Q to P , where $n_1, n_2 \geq 0$.

Each process may probe the channel to sense if its partner is ready to communicate. Probe signals $\#P$, $\#Q$ can be read and written directly without a communication protocol. A process can probe communication readiness, using 1 bit, and the data sent by its partner — n_2 bits for $\#P$, n_1 for $\#Q$. Joint $CHAN$ has one guarded command. As indicated earlier in Figure 4.2, the fat line-arrows colored red mark



$$\text{myturn}(P, Q) \wedge \text{go} \rightarrow \\ \text{myW}(P) := \text{myR}(Q) ; \text{myW}(Q) := \text{myR}(P) ; \text{yourturn}(P, Q)$$

$$\#P \equiv \{\text{myR}(Q), \text{myturn}(Q)\} \quad \#Q \equiv \{\text{myR}(P), \text{myturn}(P)\}$$

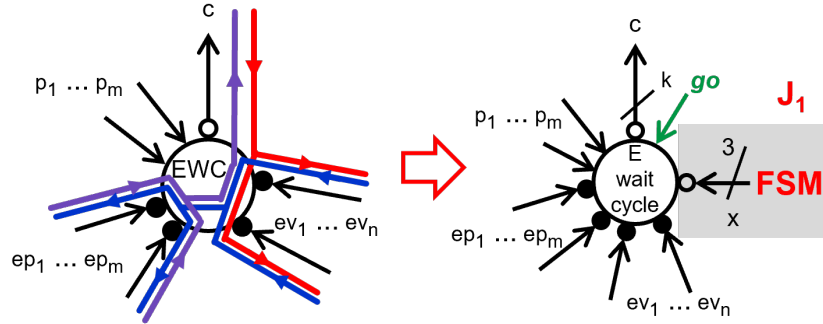
Figure 4.12: Joint CHAN library element.

the control flow in Joint CHAN. Note that the line-arrows look the same as for Joint VAR in Figure 4.2, even though VAR sequences the flow of control for r and w , while CHAN synchronizes the control flows for P and Q . The line-arrows serve as a visual aid but do not replace a Joint's guarded command specification.

Joint $E_{\text{waitcycle}}$: Joint $E_{\text{waitcycle}}$ is a special expression Joint for a nondeterministic select statement whose guards have *probes*. Joint $E_{\text{waitcycle}}$ outputs a Boolean bit list of size k representing the k number of guards in the nondeterministic select statement, where $k \geq 0$. Probes change dynamically – whenever a process is ready to communicate. Therefore, Joint $E_{\text{waitcycle}}$ waits until some guard is TRUE, which may require waiting for a probe to be TRUE. When some guard is TRUE, Joint $E_{\text{waitcycle}}$ requests for probe snapshots. A probe snapshot is a stable value of a probe captured and provided by Joint $WAIT_{\text{cycle}}$ (discussed in the next paragraph) when requested. Probe snapshots are necessary to give probes a specific time window to report if the probe would participate in the selection. Probe snapshots provide a stable evaluation of the guards for selection by Joint *SEL*.

$E_{\text{waitcycle}}$, with startup port c , uses an internal 3-bit FSM port x to sequence three guarded commands, see Figure 4.13. *Probes* come into Joint $E_{\text{waitcycle}}$ as raw signals with the formal names p_1, \dots, p_m . The raw signals come directly from a Joint *CHAN*'s probe signal ($\#P$ or $\#Q$ of Figure 4.12). The stable probe snapshots come in as protocol signals $myR(ep_i)$ at port ep_i , with index i , $0 \leq i < m$. Likewise, other non-probed expressions (variables) come in as $myR(ev_j)$ at port ev_j , with index j , $0 \leq j < n$.

But where p_i and $myR(ev_j)$ are evaluated early, $myR(ep_i)$ are evaluated after the



$\backslash\backslash$ evaluate non-probe expressions (get latest values of the variables)
 $myturn(c, ev_1, \dots, ev_n, ep_1, \dots, ep_m, x) \wedge go \wedge myR(x)[0] \rightarrow$
 $yourturn(ev_1, \dots, ev_n, x)$

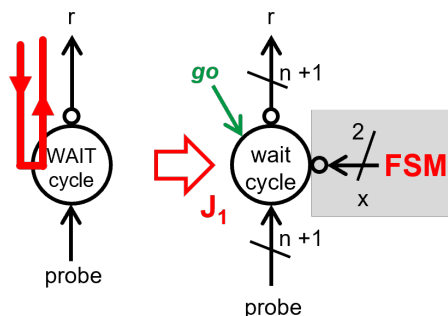
$\backslash\backslash$ wait until some guard is TRUE before taking probe snapshots
 $myturn(c, e_1, \dots, e_n, ep_1, \dots, ep_m, x) \wedge go \wedge myR(x)[1] \wedge$
 $(f'[1] \vee \dots \vee f'[k]) \rightarrow yourturn(ep_1, \dots, ep_m, x)$

$\backslash\backslash$ return stable evaluation results
 $myturn(c, e_1, \dots, e_n, ep_1, \dots, ep_m, x) \wedge go \wedge myR(x)[2] \rightarrow$
 $myW(c) := \{f[1], \dots, f[k]\}; yourturn(c, x)$

Figure 4.13: Joint $E_{\text{waitcycle}}$ library element. The specification uses $f'[i]$ to denote the evaluation of the i^{th} guard expression, where $1 \leq i \leq k$, using the variables and the raw probe signals, p_1, \dots, p_m , for probes $\#p_1$ to $\#p_m$ in the guards. The specification uses $f[i]$ to denote the evaluation of expressions using stable probe snapshots, $myR(ep_1), \dots, myR(ep_m)$, for probes $\#p_1$ to $\#p_m$ in the guards. This figure omits the bit widths for p_1 to p_m , ep_1 to ep_m , and ev_1 to ev_n to reduce visual busyness.

early evaluations have produced at least one Boolean guard bit that is TRUE. This early versus late evaluation is clearly visible in the guarded command specification in Figure 4.13.

Joint $WAIT_{cycle}$: Joint $WAIT_{cycle}$ provides a stable probe snapshot. $WAIT_{cycle}$, with startup port r , uses an internal arbiter and 2-bit FSM port x to arbitrate the raw probe input against $myturn(x)$ for a duration of one FSM cycle. $WAIT_{cycle}$ returns a communicate readiness bit and the probe data, hence the entire probe. Its guarded command specification uses mutual exclusive arbiter results $(grant_{probe}, grant_x) = arbiter(probe[0], myturn(x))$, where $probe[0]$ is the communication readiness Boolean bit of the probe. For example, $\#P = \{myR(Q), myturn(Q)\}$ for probe $\#P$ in Figure 4.12.



\\arbitrate for one FSM cycle
 $myturn(r, x) \wedge go \wedge myR(x)[0] \rightarrow yourturn(x)$

\\return stable probe snapshot
 $myturn(r, x) \wedge go \wedge myR(x)[1] \rightarrow myW(r) := (0 \text{ if } grant_x, \text{ probe if } grant_{probe}) ;$
 $yourturn(r, x)$

Figure 4.14: Joint $WAIT_{cycle}$ library element.

4.2.5 Dataflow Class - Joints COPY, SRC, SNK, FORK, RROT

This class of Joints is our implementation of dataflow or streaming nodes. These Joints are dependent on the functionality applied to streaming data. Joints *COPY*,

SRC, and *SNK* introduced in Sections 3.2 and 3.3 are examples of dataflow Joints, as are Joints *fork* and *rrot* used in Joint *FSM* in Section 4.2.2. However, this is not an exhaustive list of Joints in the dataflow class.

4.3 Compilation to Links and Joints

In our design flow, *Qnà*, the compilation approach to Link-Joint networks uses syntax-directed translation. With syntax-directed translation, our compiler walks through a program’s parse tree and emits a Joint (or a small Link-Joint network) for each program construct. It also emits Links to connect each Joint to other constructs while storing the state between them.

We can compile either dataflow or CHP programs. To illustrate the compilation to Links and Joints, we use an example, **a two-stage first-in-first-out (FIFO) buffer**. This linear FIFO buffer can store zero, one, or two data items. In Figures 4.15 and 4.16, we give two program versions of this design — an ACT *dataflow* program version and an ACT hierarchical control-flow version — and their compiled Link-Joint networks.

The ACT *dataflow* program version, *FIFO2_dataflow* in Figure 4.15, has one input channel, L , one output channel, R , and an internal channel, M . The program copies data coming over L to M and copies data coming over M to R . Our compilation for dataflow programs follows a “**store before use**” principle for the streaming channels while the channel computations become Joints. We compile this *FIFO2_dataflow* program as follows.

- a) We translate program statement $L \rightarrow M$ to Link L_{d1} to store the data coming in on program channel L , and Joint J_{d1} whose input port, *in*, connects to L_{d1} and who copies L_{d1} ’s data to its output port, *out*, for program channel M .

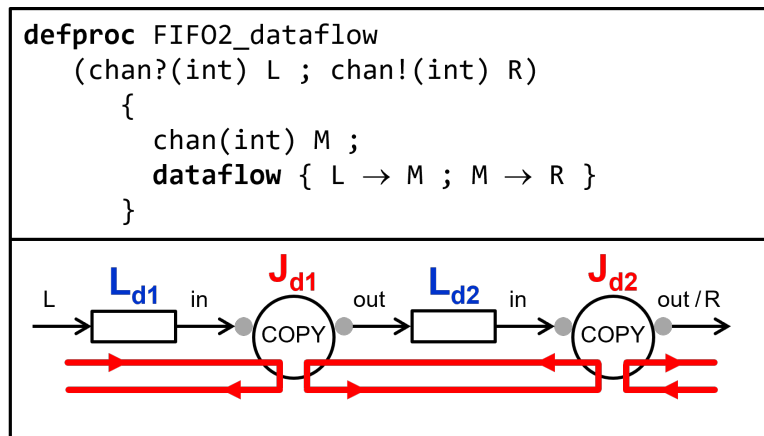


Figure 4.15: An ACT dataflow program for a two-stage FIFO buffer (top) with its corresponding compiled Link-Joint network (bottom). Data flow from left to right, coming in at Channel **L** and leaving at Channel **R**. The red line-arrows mark the flow of control in the Link-Joint network.

- b) Similarly for $M \rightarrow R$, we generate Link L_{d2} , to store the data for M before Joint J_{d2} uses L_{d2} 's data by copying the data to its output port *out* for program channel R .

The ACT hierarchical control-flow version, in Figure 4.16, has a top-level program, *FIFO2_controlflow*, with two *onebuf* instances, $b0$ and $b1$, that operate in parallel and communicate to external channels L and R and to each other. We compile *FIFO2_controlflow* into:

- a) *onebuf* Link-Joint network instances for processes $b0$ and $b1$ (the grey boxes in Figure 4.16),
- b) *PAR* Joint J_7 to execute $b0$ and $b1$ in parallel, and
- c) *CHAN* Joint J_8 to combine $b0.R$ and $b1.L$ into a single channel for internal communication between $b0$ and $b1$.

Each *onebuf* process, programmed in CHP, can store zero or one data item. We

compile *onebuf*, $*[L?x ; R!x]$, by following its syntactic structure as shown by the Joints in the grey box in Figure 4.16.

- We start with repetition “*”, and generate a repeat Joint: *REP* Joint J_1 .
- The repeated program fragment, $L?x ; R!x$, is sequential, as indicated by the

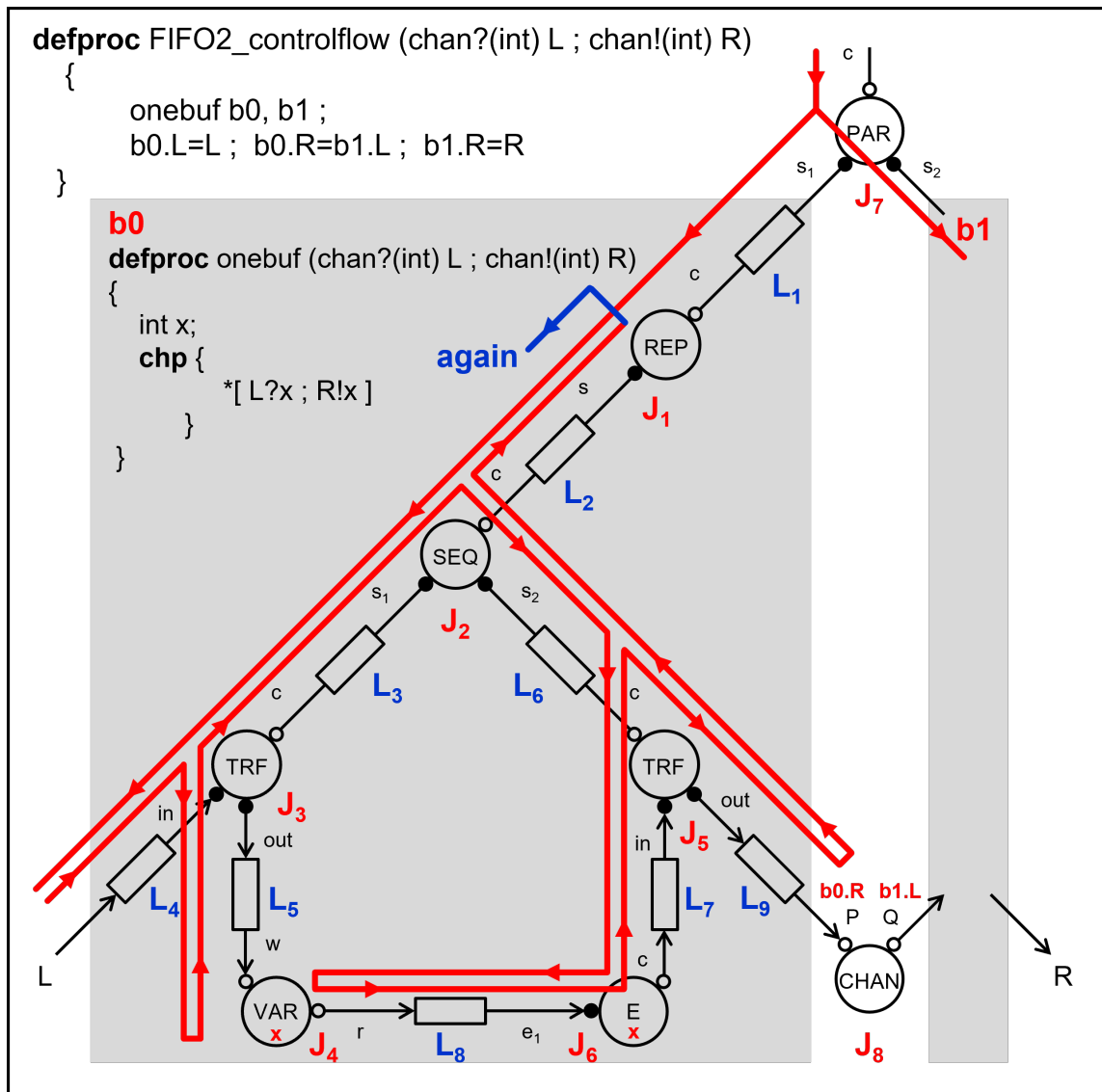


Figure 4.16: An ACT hierarchical CHP program for a two-stage FIFO with its corresponding compiled Link-Joint network. Data flow from left to right, coming in at channel *L* and leaving at channel *R*. Red and blue line-arrows mark the flow of control through the network.

sequence operator “;”, and so we generate a sequence Joint, *SEQ* Joint J_2 , and connect it to *REP* using Link L_2 .

- c) The first of the two sequenced program statements, $L?x$, is a communication input over *onebuf* channel L , which requires synchronization with a communication output over L by a parallel process. The value sent by the parallel process is stored in *onebuf* variable x . We compile $L?x$ to (1) *TRF* Joint J_3 , (2) *VAR* Joint J_4 for x , (3) Links L_4 and L_5 to connect L to x via Joint *TRF*, and (4) Link L_3 to connect this translation to *SEQ* port s_1 .
- d) The second program statement, $R!x$, is a communication output over *onebuf* channel R , which compiles to (1) *TRF* Joint J_5 , (2) *E* Joint J_6 for output expression “ x ,” (3) Links L_7 , L_8 , L_9 to connect variable x to *E* and $b0.R$ via Joint *TRF*, and (4) Link L_6 to connect this translation to *SEQ* port s_2 .

Note that for illustration, we have shown the generated Link-Joint networks graphically in Figures 4.15 and 4.16. However, our compiler generates maps of Links and Joints for each design, including any detail it obtains from the program parse tree that may be useful for our design exploration and refinement. Figure 4.17 shows a summarized output from the compiler for *FIFO2_controlflow* in Figure 4.16. These maps serve as inputs to our in-house Verilog generator tool, which produces a Verilog module (like Figure 3.17) and instantiates Link modules and Joint modules based on the maps of Links and Joints generated. We fill out a testbench using the Verilog module and simulate the Link-Joint network in Verilog.

```

1  /******  

2  /* Process <proc_name, [(CHP_name, actual_portname)]> */  

3  Process < onebuf, [(c, 7A),(L, 10B),(R, 13B)]>  

4  

5  /*Link < l_id, bw_varA2B, bw_varB2A, j_A, j_B, CHP_chan>*/  

6  Link < 1, 0, 0, -1, 1, c >  

7  Link < 2, 0, 0, 1, 2, >  

8  Link < 3, 0, 0, 2, 3, >  

9  Link < 4, 0, 32, 3, -1, L >  

10 Link < 5, 32, 0, 3, 4, >  

11 Link < 6, 0, 0, 2, 5, >  

12 Link < 7, 0, 32, 5, 6, >  

13 Link < 8, 0, 32, 6, 4, >  

14 Link < 9, 32, 0, 5, -1, R >  

15  

16 /* Joint < j_id, type, map_port_names, {extra_info} >*/  

17 Joint < 1, Rep, [(c,1B),(s,2A)], {...} >  

18 Joint < 2, Seq, [(c,2B),(s_1,3A),(s_2,6A)], {...} >  

19 Joint < 3, Trf, [(c,3B),(in,4A),(out,5A)], {...} >  

20 Joint < 4, Var, [(r,8B),(w,5B)], {CHP_var: 'x', bw: 32,  

    ...} >  

21 Joint < 5, Trf, [(c,6B),(in,7A),(out,9A)], {...} >  

22 Joint < 6, E , [(c,7B),(e_1,8A)], {expr: 'x', var2port:(x  

    ,e_1), bw:32, ...} >  

23  

24 /******  

25 /* Process <proc_name, [(CHP_name, actual_portname)]> */  

26 Process < FIFO2, [(L, 8.L),(R, 7.R),(c, 5A)]>  

27  

28 /*Link < l_id, bw_varA2B, bw_varB2A, j_A, j_B, CHP_chan>*/  

29 Link < 1, 0, 0, -1, 4, c >  

30  

31 /* Joint < j_id, type, map_port_names, {extra_info} >*/  

32 Joint < 1, onebuf<>, [], {CHP_inst: 'b0', port_alias: [(L  

    ,L),(R,8.P),(c,7.s_1)], ...} >  

33 Joint < 2, onebuf<>, [], {CHP_inst: 'b1', port_alias: [(L  

    ,8.Q),(R,R),(c,7.s_2)], ...} >  

34 Joint < 8, Chan, [(P,1.R),(Q,2.L)], {...} >  

35 Joint < 7, Par, [(c,1B),(s_1,1.c),(s_2,2.c)], {...} >

```

Figure 4.17: Our compiler generated output for the two-stage FIFO example in Figure 4.16, only including network topology information and excluding other details.

4.3.1 ACT Compiler Modifications

As mentioned in the introduction of this chapter, we compile ACT programs into Link-Joint networks. The Yale ACT ecosystem has a tool, *CHP2PRS* [33], that compiles CHP programs into *Production Rule Set (PRS)* netlists. PRS [40] is a gate-level specification language. We chose to adapt the tool, reprogramming the compiler to produce Link-Joint networks instead. We kept the parser and syntax checker but modified some compilation rules and internal data structures. We kept the implementation of the compiler simple and avoided premature optimizations to keep the Link-Joint network simple, functional, and general.

Some implementation decisions made for our compiler include:

1. Restrict computation to Joints and push out storage into connected Links.
2. Do not expand expressions. Instead of emitting a Link-Joint network for the parse tree of an expression, emit a Joint that computes the entire expression with Link or probe connections to any variables or probes used in the expression.
3. Refrain from making premature design choices during compilation. For example, we assume all Links store data and make the decision to omit data storage at a later refinement stage.
4. In relation to item 3, compile self-assignments, like $x := x + 1$, without the traditional introduction of auxiliary variables and assignments. Instead, in our later refinement process, we ensure that there is intermediate storage on the assignment path between ports r and w of Joint *VAR* for x , to separate the old and new value of x for the duration of the assignment.
5. For CHP program constructs with conditional expressions (guards) like *SELECT* and *LOOP* statements, combine all the guards into a single expression.

The result of the single expression is a bit vector containing the Boolean values of the corresponding guards.

6. For hierarchical programs, generate explicit Links and Joints to connect parallel processes in the program. For example, Figure 4.16 shows that Joints *PAR* and *CHAN* and corresponding Links are generated to connect the two instances of process *onebuf*, *b0* and *b1*.
7. Refrain from indicating the network’s communication protocols and data representation.
8. Remove redundant components in the network after the translation is done.
9. Provide mutual exclusive multiplexed access to shared resources such as variables and channels.
10. Remember initial state information for the regular operation of the compiled components, but avoid baking it into the compiled Link-Joint network to allow different initialization settings for other evaluation purposes – see Section 3.3.

We can summarize the compilation rules based on the CHP grammar and our compiler implementation decisions as follows. Note that we use functions such as *genLink()*, to emit a Link, *emit_Joint()*, to emit a specific Joint type, *connect_mux()*, to connect to a multiplexer of a variable or channel. Joints for CHP statements and expressions have startup links to give control that triggers an action in the Joint.

```

Variables: [v] ⇒
    let r_id = genLink(), w_id = genLink() in
    emit_VAR(bit_width, "v", r_id, w_id)
    emit_MUX(bit_width, "v", r_id, "RMUX")
    emit_MUX(bit_width, "v", r_id, "WMUX")

```

Channels: $[A] \Rightarrow$

```

let p_id = genLink(), q_id = genLink() in
emit_CHAN(bit_width, "A", p_id, q_id)
emit_MUX(bit_width, "A", p_id, "RMUX")
emit_MUX(bit_width, "A", q_id, "WMUX")

```

Expression: $[E] \Rightarrow$

```

let startup = genLink(),
    lst = [connect_mux(genLink(), v, "RMUX")]
foreach v in E in
emit_E(startup, bit_width, E, lst)

```

CHP Statements:

$[skip] \Rightarrow$ **let** startup = genLink() **in** emit_SKIP(startup)

$[v := E] \Rightarrow$ **let** startup = genLink(), in_id = [E].startup,
out_id = connect_mux(genLink(), v, "WMUX") **in**
emit_TRF(startup, bit_width, in_id, out_id)

$[A ! E] \Rightarrow$ **let** startup = genLink(), in_id = [E].startup,
out_id = connect_mux(genLink(), A, "WMUX") **in**
emit_TRF(startup, bit_width, in_id, out_id)

$[A ? v] \Rightarrow$ **let** startup = genLink(),
in_id = connect_mux(genLink(), A, "RMUX"),
out_id = connect_mux(genLink(), v, "WMUX") **in**
emit_TRF(startup, bit_width, in_id, out_id)

$[S_1 ; \dots ; S_n] \Rightarrow$ **let** startup = genLink(),
s_id = [[S_i].startup **for** i = 1 **to** n] **in**
emit_SEQ(startup, s_id)

$[S_1 , \dots , S_n] \Rightarrow$ **let** startup = genLink(),
s_id = [[S_i].startup **for** i = 1 **to** n] **in**
emit_PAR(startup, s_id)

$[[G_1 \rightarrow S_1 [] \dots [] G_n \rightarrow S_n]] \Rightarrow$
let startup = genLink(), E = {G₁, ..., G_n},
g_id = [E].startup,
s_id = [[S_i].startup **for** i = 1 **to** n] **in**
emit_SELdet(startup, g_id , s_id)

```

[[G1 → S1 [] ... [] Gn → Sn [] else → Sn+1]] ⇒
  let startup = genLink(), E = {G1, ..., Gn},
      g_id = [E].startup,
      s_id = [[Si].startup for i = 1 to n+1] in
  emit_SELdet(startup, g_id, s_id)

[[|G1 → S1 [] ... [] Gn → Sn|]] ⇒
  let startup = genLink(), E = {G1, ..., Gn},
      g_id = [E].startup,
      s_id = [[Si].startup for i = 1 to n] in
  emit_SELnondet(startup, g_id, s_id)

[[|G1 → S1 [] ... [] Gn → Sn [] else → Sn+1|]] ⇒
  let startup = genLink(), E = {G1, ..., Gn},
      g_id = [E].startup,
      s_id = [[Si].startup for i = 1 to n+1] in
  emit_SELnondet(startup, g_id, s_id)

[*[G1 → S1 [] ... [] Gn → Sn]] ⇒
  let startup = genLink(), E = {G1, ..., Gn},
      g_id = [E].startup,
      s_id = [[Si].startup for i = 1 to n] in
  emit_REP(startup, g_id, s_id)

```

4.3.2 Compiler Optimization

Typical syntax-directed compilation is quite transparent, giving the designer full design control. Therefore, the generated circuit is only as efficient as the program given at the beginning of the design process. Brunvand [4], in his design flow, made circuit improvements (optimizations) a post-compilation step. The Tangram design flow provided post-compilation steps for peephole optimizations [49].

Our compilation is equally transparent. The compiler in *Qnà* supports post-compilation steps to further optimize compiled Link-Joint networks and gate-level circuits. Recall that our philosophy is to make design decisions as late as possible. *Qnà* is already set up with a step-wise post-compilation refinement process (Chapter 5) for lower-level design decisions. Typical optimizations that are better addressed

after compilation are local or peephole optimizations on the Link-Joint netlist and peephole optimizations at the circuit level.

Simulation, test, and debug at the language level can provide valuable information for pre-compilation program-level optimizations. Likewise, simulation, test, and debug at a combination of language and Link-Joint levels can give further details for post-compilation refinements and peephole optimizations at the Link-Joint and circuit levels. Chapter 5 provides more details on this topic.

4.4 Chapter Contributions

Though the compilation approach and part of the Yale *CHP2PRS* compiler existed before our research, our work focuses on translating high-level programs into Links and Joints. The work on the compiler in *Qnà* and the Link-Joint library elements are joint efforts between my supervisor and me. The following contributions are largely mine:

- I re-engineered the *CHP2PRS* compiler to generate circuit-neutral Link-Joint networks from CHP programs instead of gate-level PRS netlists. This modification comprised the creation of additional data structures, including those for Links and Joints, maintenance of these structures during compilation, and some differences in the compilation scheme. This re-engineering effort contributed to an entire refactoring of the *CHP2PRS* codebase [33]. In addition, I showed how to translate ACT dataflow programs into Link-Joint networks.
- I specified the Link-Joint library elements with the shared variable semantics. Most of the elements had guarded command specifications using the previous Links and Joints semantics as seen in [60, 62]. The new specifications now (1) fully incorporate Links, (2) completely define a Joint's internal flow control using

the generic *FSM* model of Section 4.2.2 and (3) support hierarchical Link-Joint networks.

- I developed an in-house tool, a Python script that produces a Verilog module instantiating Link modules and Joint modules from the maps of Links and Joints generated by the compiler. The script also produces (1) any extra Joint module that needs to be created on the fly, such as a Joint module for an expression, and (2) an associated test bench. Currently, we run the Verilog modules on Icarus Verilog [76], and we view the simulated results as waveforms using GTKwave [7] – both of which are freely available.

Chapter 5: Refinement

A Link-Joint network is an abstract representation of a design, but its specification gives enough detail to exhibit the design’s functionality and asynchronous behavior. Instead of deciding all circuit-related implementation choices before compiling as done in related design flows [1–6, 32, 33, 49, 50, 72], we layer these choices as refinements of the Link-Joint network. *Refinement is a process of step-wise decisions about implementation options for a Link-Joint network.* Starting with a circuit-neutral Link-Joint network, the refinement process ends with a circuit; see Figure 5.1.

This chapter showcases Link-Joint network refinement examples, some gate-level implementations, and details on how we preserve the relation of each refinement to the original program. Some refinement details discussed in this chapter were presented and published in a paper [17] at the 2023 ASYNC conference. Our refinement process aligns with the Link-Joint property of binding decisions as late as possible. Refinement provides the opportunity to explore the variety of implementation styles available. This opportunity is essential because most designers are prone to sticking with implementation choices based on their familiarity and experience.

Links and Joints make refinement easy and feasible because many implementation choices are local to either Links or Joints and are usually independent of the final asynchronous circuit family or fabric used. This chapter does not recommend an ideal mix or set of refinements because that choice would depend on the design’s specific requirements and constraints. Instead, this chapter showcases the Link-Joint model as a vehicle for exploring and executing choices made.

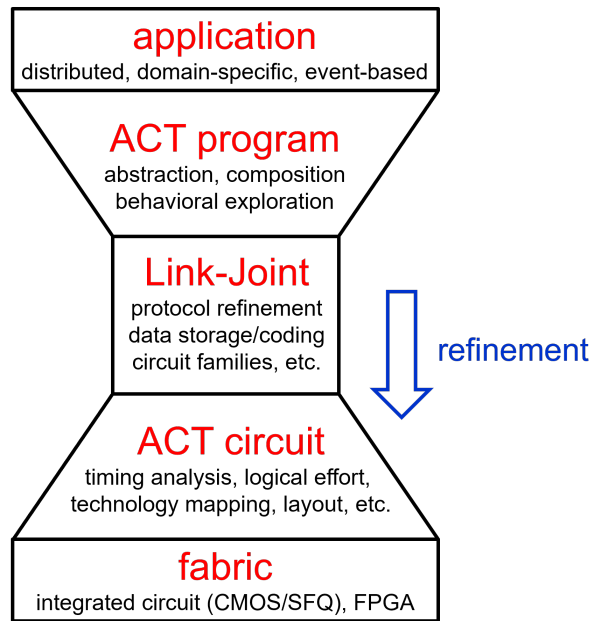


Figure 5.1: Refinement as a focus in the $Qnà$ design flow.

5.1 Asynchronous Design Styles

Asynchronous circuits come in different styles due to various design considerations, trade-offs, specialized applications, evolution, and innovation. These variations are dependent on implementation decisions made in the design process. We consider these decisions as a large buffet of options open to the circuit designer to use and explore. Some refinement options considered in the context of this dissertation include the following:

1. Handshakes, e.g., 2-phase and 4-phase
2. Signaling logic, e.g., level logic, pulse logic, transition logic
3. Data encoding, e.g., bundled (single-rail) or dual-rail data encoding
4. Guard selection in nondeterminism, e.g., round robin, arbitration
5. Data storage, e.g., where and where not to store

6. Test extensions, e.g., throughput counters, state to control and observe
7. Asynchronous circuit family, e.g., Click, Set-Reset, GasP, RSFQ, Mousetrap, Micropipelines

Some of the above options have been explained in Sections 2.1 and 2.2, while others will be discussed as they are used in the rest of this dissertation.

5.2 Refinement Examples

The following subsections provide specific examples of refining Link-Joint networks. Note that there may be several refinement steps before generating gate-level descriptions. Each intermediate refinement remains a Link-Joint network until it reaches the gate level.

5.2.1 Data Storage Refinement: To Store or Not to Store

Parallel operations, especially quasi-delay-insensitive ones, require data to be stored between the sender and the receiver. The default representation of Links stores all transported data bits. This choice can be expensive in terms of delay, area, and power when there are many data bits. Therefore, this section explores data storage elimination refinement to save circuit area and power while maintaining the delay-insensitivity of the data exchange.

To illustrate, we use the bidirectional communication in Figure 5.2, compiled from the parallel program fragments $\text{chp}\{\dots p?x_1!y_1\dots\}$ and $\text{chp}\{\dots p!y_2?x_2\dots\}$, which exchange y_1 , y_2 values and store these locally in x_2 , x_1 . The fat line arrows in the figure mark the control flow in the compiled communication statement for the leftmost process. CHP translations often lead to a path behavior where an earlier Link stores

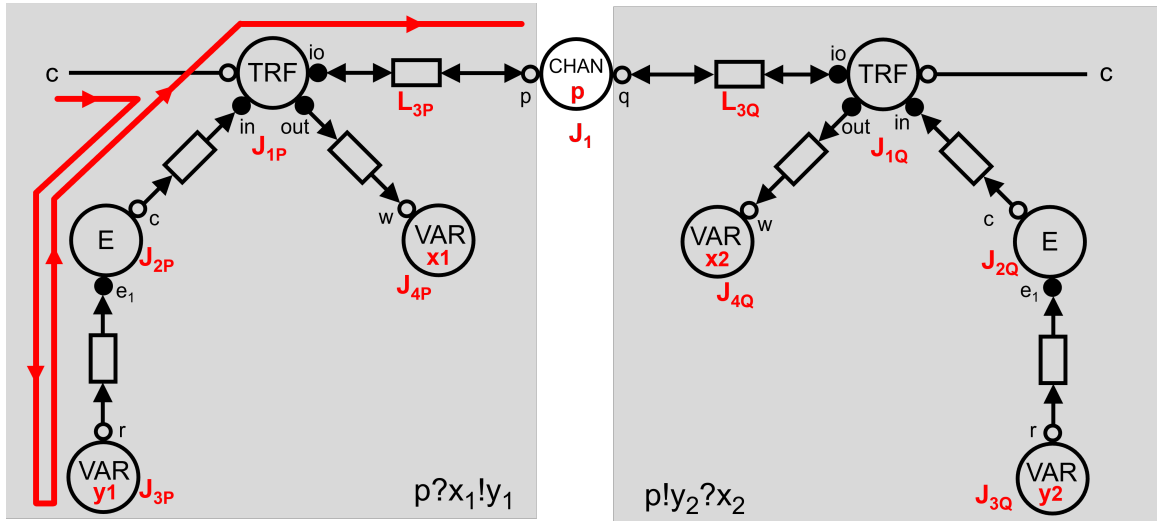


Figure 5.2: Link-Joint network representing two parallel program fragments that exchange data between both sides.

data for a later Link in the path [62]. In this example, the internal Link in Joint *VAR* stores the value of y_1 , which is not used until the data exchange at *CHAN p*; thus, the Links between them do not need to store the value of y_1 . With this behavior, all Links between data holders and users can simply transport data values without storing them.

We present three solutions to address the data storage elimination for the example in figure 5.2. Both sides of the communication follow similar execution paths.

Solution 1: Keep the internal storage in both Joint *VAR* y_1 and Joint *VAR* y_2 ; store values received after data exchange by Joint *CHAN p* in Links L_{3P} and L_{3Q} ; and eliminate other (non-internal) data storage otherwise. Links in this solution use 2-phase handshaking. We depict the execution path of the example with this solution strategy in Figure 5.3 with thick red and blue lines.

The execution path for the left side of this solution starts at port c of J_{1P} , continues via J_{2P} to port r of J_{3P} to read the value of program variable y_1 , which it sends via J_{2P}

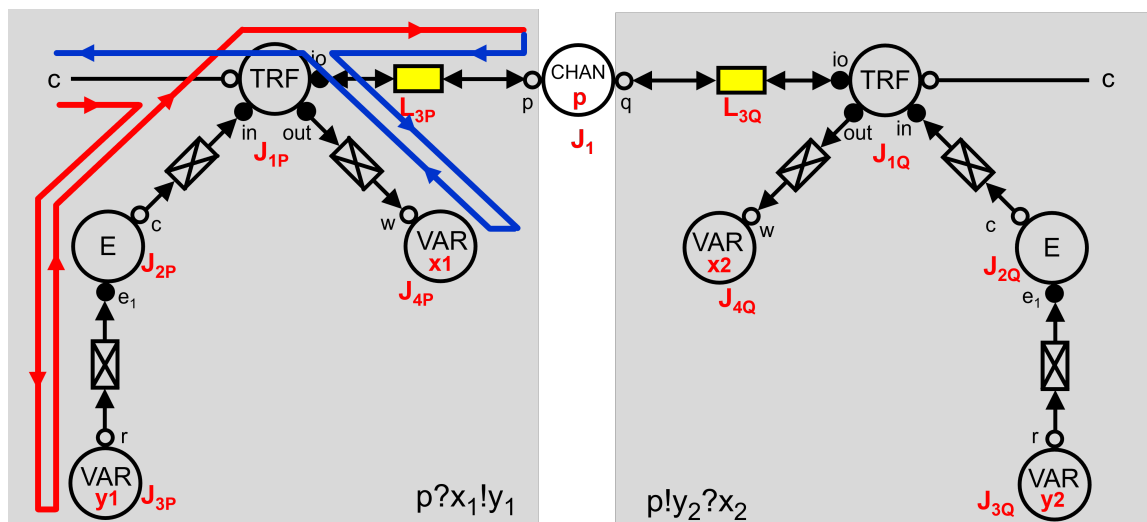


Figure 5.3: Link-Joint network showing the first refinement solution for example in Figure 5.2. We use rectangles with a cross for Links without data storage and rectangles colored yellow for Links that store data coming from CHAN, L_{3Q} and L_{3P} .

and J_{1P} to CHAN port p , where it stalls until the communication partner at port q is ready to exchange its value of y_2 for the value of y_1 . When ready, the path continues by storing the received y_2 value into the L_{3P} storage location for data from J_1 . The execution path for the right side is symmetric to that of the left.

After the data exchange by CHAN, the values of y_1 and y_2 are stored in Links L_{3Q} and L_{3P} respectively. Therefore, each process, specifically Joints J_{1P} and J_{1Q} , can independently and *at their own pace* transfer the received values and now locally store them into VAR x_1 Joint J_{4P} and VAR x_2 Joint J_{4Q} . By storing exchanged data in Links L_{3P} and L_{3Q} , we ensure that the original values are not overwritten by the start of a new communication.

Note that storing the data values and relinquishing both Link turns for L_{3P} and L_{3Q} altogether take one atomic CHAN action. This atomic CHAN action enables both partners to finish their execution paths independently without retarding each other. The guarded command specification for Joint TRF used here **extends** that

of Joint *TRF* in Section 4.2.1 and Figure 4.4 as follows:

$$\begin{aligned}
 & \text{myturn}(c, \text{in}, \text{io}, \text{out}, x) \wedge \text{go} \rightarrow \\
 & \text{myR}(x)[0] \rightarrow \text{yourturn}(\text{in}, x) \\
 & \text{myR}(x)[1] \rightarrow \text{myW}(\text{io} := \text{myR}(\text{in}); \text{yourturn}(\text{io}, x)) \\
 & \text{myR}(x)[2] \rightarrow \text{myW}(\text{out} := \text{myR}(\text{io}); \text{yourturn}(\text{out}, x)) \\
 & \text{myR}(x)[3] \rightarrow \text{yourturn}(c, x)
 \end{aligned}$$

Solution 2: Store only internal Link data for both Joint *VAR* y_1 and Joint *VAR* y_2 , and eliminate data storage in all other (non-internal) Links. With this solution strategy, *VAR* y_1 **must** hold its data, without the value changing, until the peer process has copied the value into *VAR* x_2 Joint J_{4Q} . We depict the corresponding execution path with thick red and blue lines in Figure 5.4.

Execution in this refinement solution follows the same path in Figure 5.3 except the execution of the process on the left is **not** independent of the execution on the right. The right process must stall any task following $p!y_2?x_2$ until the *now* L_{3P}

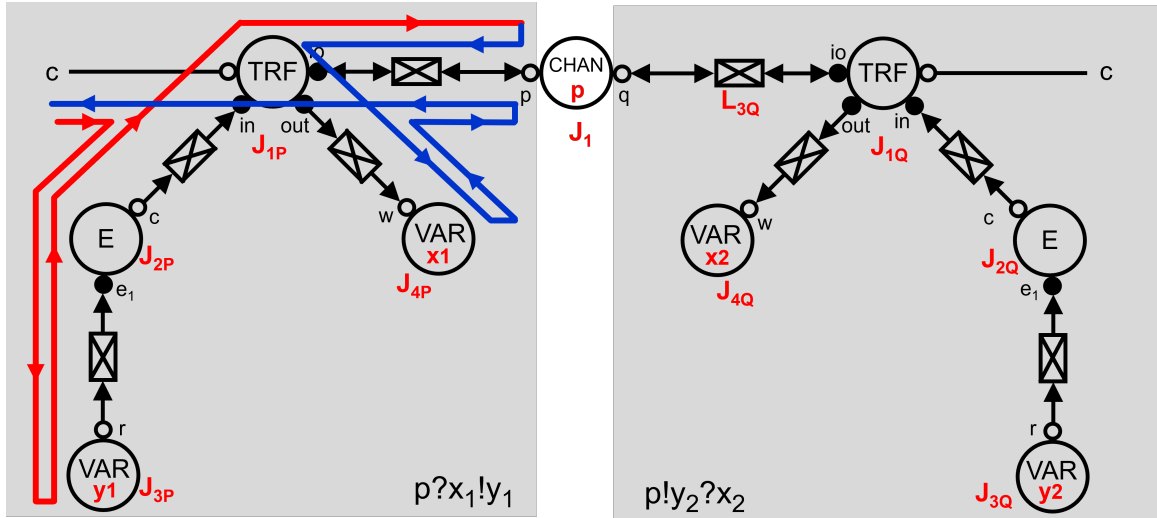


Figure 5.4: Link-Joint network showing the second refinement solution, for example, in Figure 5.2. We use rectangles with a cross for Links without data storage, including L_{3Q} and L_{3P} .

un-stored y_2 value is written into J_{4P} (for x_1) and likewise the left process must stall any task following $p?x_1!y_1$ until the *now* L_{3Q} *un-stored* y_1 value is written into J_{4Q} (for x_2). The control flow of the processes now includes an extra TRF to CHAN trip after writing $VAR x_1$ and $VAR x_2$ respectively, to report that the data have been written, and the communication peer can thus stop holding that data. Relinquishing both turns for L_{3P} and L_{3Q} during the second trip is an atomic *CHAN* action, after which both partners can continue the rest of their executions independently.

The execution path extension in Figure 5.4 leads to a 4-phase handshake over Links L_{3P} and L_{3Q} . Because each process holds the data it sends to *CHAN* throughout the communication, we can implement each 4-phase communication as twin 2-phase communications. The persistence of data, from the moment data arrive to communication completion at each *TRF* port c , extends the earlier *TRF* specification in Figure 5.3 as follows:

$$\begin{aligned}
& myturn(c, in, io, out, x) \wedge go \rightarrow \\
& \quad myR(x)[0] \rightarrow yourturn(in, x) \\
& \quad myR(x)[1] \rightarrow myW(io) := myR(in); yourturn(io, x) \\
& \quad myR(x)[2] \rightarrow myW(out) := myR(io); yourturn(out, x) \\
& \quad myR(x)[3] \rightarrow yourturn(io, x) \\
& \quad myR(x)[4] \rightarrow yourturn(c, x)
\end{aligned}$$

Solution 3: Like **Solution 1**, the Links in this refined network use 2-phase handshaking only. Also, like **Solution 2**, this refined network stores only internal Link data. However, there are constraints on the network, unlike in solutions 1 and 2. The corresponding constraints are *delay-sensitive* and can be expressed as relative timing constraints [67] within the communication network or its surroundings, as follows.

- *static constraint*: From the moment *CHAN* exchanges data, the leftmost process and its peer must write the data they receive in *VAR* x_1 and x_2 , respectively, before completing their communication parts at *TRF* port c .
- *dynamic constraint*: In reality, it suffices that the leftmost process writes x_1 before y_2 changes reach x_1 . Though this dynamic constraint may be more challenging to analyze, it may point to alternative static constraints.

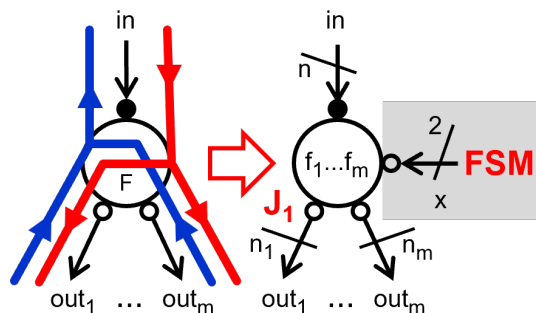
As a guideline for selecting one out of the three refinement solutions: **Solution 1** is robust and fast, **Solution 2** is robust and small, and **Solution 3** is both fast and small, but timed. Each solution may replace Links with data storage by Links without, and swap in different *TRF* Joints, but *none of these replacements change the topology of the compiled Link-Joint network and its syntax-directed relation to the program!*

Note that data storage elimination is an optimization with test consequences. Chapter 6 will discuss the necessary requirements to control actions and initialize or observe state. Eliminating data storage in Links reduces opportunities for testing and debugging. Therefore, this data storage refinement must be done with test and debug opportunities in mind.

5.2.2 Atomicity Refinement: Fuse or Split Joint Atomicity

Communication protocols and Joints can provide atomicity when and where needed, for as long as needed. Figures 5.3 and 5.4 demonstrate the “for as long as needed” provision: the control flow marked by the fat line-arrows shows each network operation as if it were a single atomic action — which makes sense as the network corresponds to a single communication statement in the program. This section looks at the other end of the spectrum: splitting an atomic Joint action into smaller actions. We are

motivated to split an atomic Joint action to facilitate its circuit implementation.



$$\begin{aligned}
 & \text{myturn}(in, out_1..out_m, x) \wedge go \wedge \text{myR}(x)[0] \rightarrow \text{yourturn}(in, x) \\
 & \text{myturn}(in, out_1..out_m, x) \wedge go \wedge \text{myR}(x)[1] \rightarrow \\
 & \quad \text{myW}(out_1) := f_1(\text{myR}(in)) ; \quad \dots ; \quad \text{myW}(out_m) := f_m(\text{myR}(in)) ; \\
 & \quad \text{yourturn}(out_1..out_m, x)
 \end{aligned}$$

Figure 5.5: Joint F library element.

Given that each atomic action corresponds to one guarded command in the specification of the Joint, we will focus on splitting guarded commands. We use Joint F in Figure 5.5 to demonstrate. F is a “heavy” Joint, with many output assignments in a single guarded command. We can make F “less heavy” by splitting its assignments into separate guarded commands while maintaining the overall command sequence. Expressing this requires refining the original Link-Joint semantics specified in Section 3.2.

First, we split a guarded command into operations performed over the same Link port. This split produces a set of guarded commands, each with fewer operations than the original. To maintain the original “fused” command sequence, we add a guarded command that relinquishes the turns on the Link ports and is executed upon completion of the guarded commands in the set.

To support this strategy, we extend the original shared variable model [16] specified in Section 3.2. The Link variable *turn* now tracks how far along Link port p is in

its command execution; everything else in the shared variable model stays the same. We extend the terminology used in the guarded command specifications accordingly.

- Boolean $myturn(p)$ is *TRUE* if and only if p has the turn but has not yet completed its command.
- As before, $myR(p)$ and $myW(p)$ are data read and written by p — going from Link variables to p and vice versa.
- Boolean $midturn(p)$ is *TRUE* if and only if p has the turn *and* completed its command.
- Assignment $halfturn(p)$ changes Link variable $turn$ so that $myturn(p)$ becomes *FALSE* and $midturn(p)$ becomes *TRUE*.
- Assignment $yourturn(p)$ changes $turn$ so $myturn(p)$ and $midturn(p)$ become *FALSE* and $myturn(p_{peer})$ becomes *TRUE*, where p_{peer} is A if p connects to B , otherwise B .

This refinement leads to 4-phase handshaking on port p with the following state-action phases: $myturn(p); halfturn(p); midturn(p); yourturn(p)$. Its peer p_{peer} sees only state $myturn(p)$ and action $yourturn(p)$ and can freely be 2-phase or 4-phase. *Joint splitting is immaterial to neighboring Joints!*

We also extend the terminology for go , the external Boolean signal for initialization, test, and debug, which is part of the guard in a guarded command and is arbitrated to permit or deny command execution. The arbitration, implicit in the original specifications, becomes visible when we split a guarded command. Because a guarded command is atomic, its execution, once started, must be completed. To model atomic executions of “heavy” guarded commands with a set of multiple “less heavy” guarded commands, we add the following terms.

- Boolean $mid(go)$ is *TRUE* if and only if the arbiter has made a decision and decided to permit command execution.
- Assignment $half(go)$ makes $mid(go)$ *TRUE*.
- Assignment $your(go)$ makes $mid(go)$ *FALSE*.

The resulting Link-Joint model is backward compatible with the original model [16] in Section 3.2.

Now, we can create and specify a “less heavy” version of Joint F in Figure 5.5. To do this, we split the two atomic guarded commands of F using the strategy described in the text below Figure 5.5, and i, j to index the m output Links of F , where $i \neq j \wedge 1 \leq i, j \leq m$.

Split 1st guarded command in Figure 5.5 — for $myR(x)[0]$

$$myturn(in) \wedge myturn(out_1..out_m) \wedge myR(x)[0] \wedge (myturn(x) \vee midturn(x))$$

$$\wedge (go \vee mid(go)) \rightarrow halfturn(in) ; half(go)$$

$$myturn(x) \wedge myturn(out_1..out_m) \wedge myR(x)[0] \wedge (myturn(in) \vee midturn(in))$$

$$\wedge (go \vee mid(go)) \rightarrow halfturn(x) ; half(go)$$

$$midturn(in, x) \rightarrow yourturn(in, x) ; your(go)$$

Split 2nd guarded command in Figure 5.5 — for $myR(x)[1]$

$$myturn(in) \wedge myturn(out_i) \wedge myR(x)[1] \wedge (myturn(x) \vee midturn(x))$$

$$\wedge (go \vee mid(go)) \wedge \bigwedge_{j=1..m}^{j \neq i} (myturn(out_j) \vee midturn(out_j))$$

$$\rightarrow myW(out_i) := f_i(myR(in)) ; halfturn(out_i) ; half(go)$$

$$myturn(x) \wedge myturn(in) \wedge myR(x)[1] \wedge (go \vee mid(go))$$

$$\wedge \bigwedge_{j=1..m} (myturn(in) \vee midturn(in)) \rightarrow halfturn(x) ; half(go)$$

$$midturn(out_1..out_m, x) \rightarrow yourturn(out_1..out_m, x) ; your(go)$$

Because the go arbiter is released and each Link $turn$ relinquished upon completion of *all* split commands, both the 1st and 2nd guarded command executions are atomic.

As a result, when the environment stops F by making go $FALSE$, it stops F safely — before, between, or after the 1st and 2nd guarded commands.

5.2.3 Nondeterministic Selection Implementation

As discussed in Section 4.2.3, nondeterministic select statements can have more than one guard $TRUE$ at a time. Our compilation approach does not specify which of the statements of the $TRUE$ guards is executed nor how the selected guard is determined. This compiler implementation is because the implementation is up to the designer, and the guard choice does not impede the correct functional behavior of the Joint as long as one guard is picked. Through refinement, we can make explicit how the selected guard is decided.

Some guard selection implementations include arbitration, round-robin, and static priority. An arbitration implementation requires an arbiter circuit connected to the guard expressions to determine which $TRUE$ guard to select. However, we can view

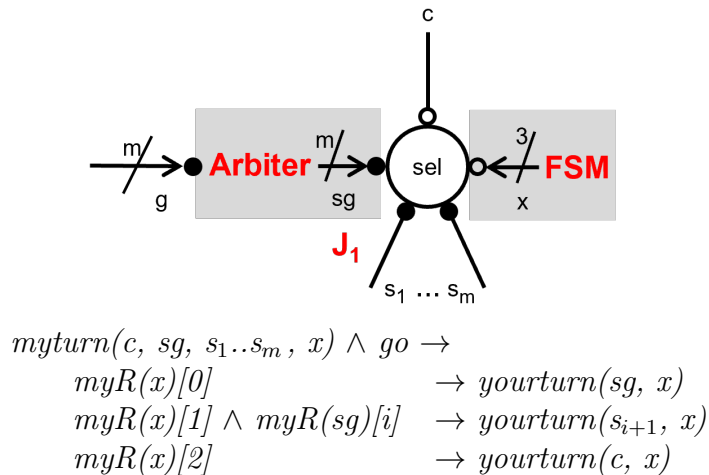
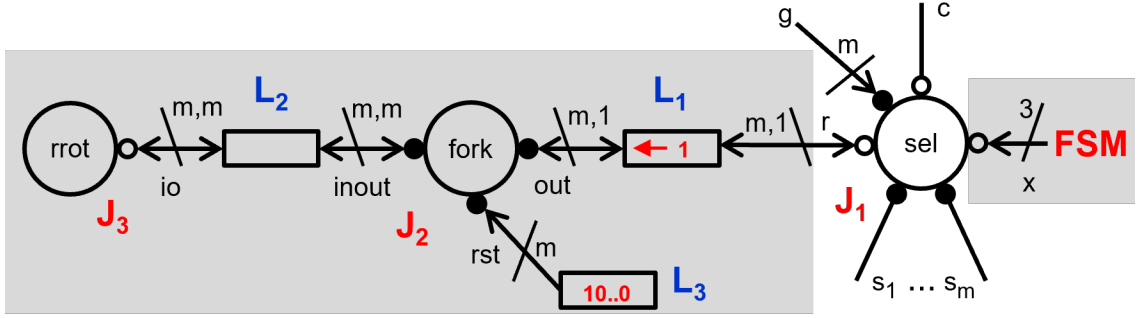


Figure 5.6: Refined Joint SEL library element representing SEL_{nondet} with an arbiter guard selection implementation.



(sel)

$$\begin{aligned}
 & \text{myturn}(c, g, r, rst, s_1..s_m, x) \wedge go \rightarrow \\
 & \quad \text{myR}(x)[0] \rightarrow \text{yourturn}(g, x) \\
 & \quad \text{myR}(x)[1] \wedge \text{myR}(r)[k] \wedge g[k] \rightarrow \text{myW}(r) := 1; \text{yourturn}(s_{k+1}, r, x) \\
 & \quad \text{myR}(x)[1] \wedge \text{myR}(r)[k] \wedge (g[0] \vee \dots \vee g[m-1]) \wedge \neg g[k] \\
 & \quad \quad \rightarrow \text{myW}(r) := 0; \text{yourturn}(r) \\
 & \quad \text{myR}(x)[2] \rightarrow \text{yourturn}(c, x)
 \end{aligned}$$

(fork)

$$\begin{aligned}
 & \text{myturn}(inout, out, rst) \wedge go \rightarrow \\
 & \quad \text{myR}(r) \rightarrow \text{myW}(inout, out) := \text{myR}(rst); \text{yourturn}(inout, out) \\
 & \quad \neg \text{myR}(r) \rightarrow \text{myW}(inout, out) := \text{myR}(inout); \text{yourturn}(inout, out)
 \end{aligned}$$

Figure 5.8: Refined Joint SEL library element representing SEL_{nondet} with a static or fixed priority guard selection implementation. Data width $m, 1$ for data between Joints *fork* and *sel* indicates that m bits go from Joint *fork* to Joint *sel* and 1 bit from Joint *sel* to Joint *fork*.

The static or fixed priority nondeterministic guard selection implementation in Figure 5.8 can be a specialization of the round-robin implementation given in Figure 5.7. The static priority implementation requires resetting the state of k to a starting 1-hot m -bit vector. The port list of the refined sel_{nondet} Joint remains the same as that of the round-robin implementation. However, there is a difference in the guarded command specification. The Joint now has to direct Joint *fork* to reset the 1-hot m -bit vector after choosing a TRUE guard. We make Link L_1 bidirectional and add an internal Link L_3 to Joint *fork* to store the starting 1-hot m -bit vector needed at reset. This allows us to initialize the Link-Joint *rrot-fork* network differently. Rather

than initializing the starting vector in Link L_2 , as in Figure 5.7, we make $myR(out) := 1$, triggering Joint *fork* to get the starting vector from Link L_3 .

5.3 Abstract Gate-level Implementations

This section presents Link-Joint abstract gate-level implementations. Abstract in this context means we are not providing the timing constraints required for the regular operation of these circuits. The implementations are done in asynchronous families: *Click* [50], *Set-Reset* [60], and *GasP* [68]. Other families follow later. *RSFQ* [27, 57] is introduced and discussed in Chapter 6, while *Micropipelines* [69] and *Mousetrap* [65] are presented in Chapter 8. At the gate level, data and *turn* pass back and forth between Link ports A and B by raising and lowering interface signals while maintaining the atomicity of guarded commands. The implementations given in the following subsections are done in Verilog and follow the approach we introduce further in Chapter 7.

5.3.1 2-phase level-signaling bundled data Link circuits

We present Link circuits for *2-phase level-signaling bundled data* circuit implementations done in three circuit families: Click, Set-Reset, and GasP. These circuits are equivalent to the Link circuits presented in 2015 [60] but use the semantics presented in Chapter 3. The relation between guarded command specifications and circuit implementations for *2-phase* port communication is as follows.

- The Boolean specification terms $myturn(p)$, $myR(p)$, $myW(p)$ become circuit signals $myturn(p)$, $myR(p)$, and $myW(p)$.
- The specification assignment term $yourturn(p)$ has circuit representation

$yourturn(p)\uparrow; ((myturn(p)\downarrow; yourturn(p)\downarrow), myturn(p_{peer})\uparrow)$ — where p_{peer} is A if p connects to Link port B , otherwise B .

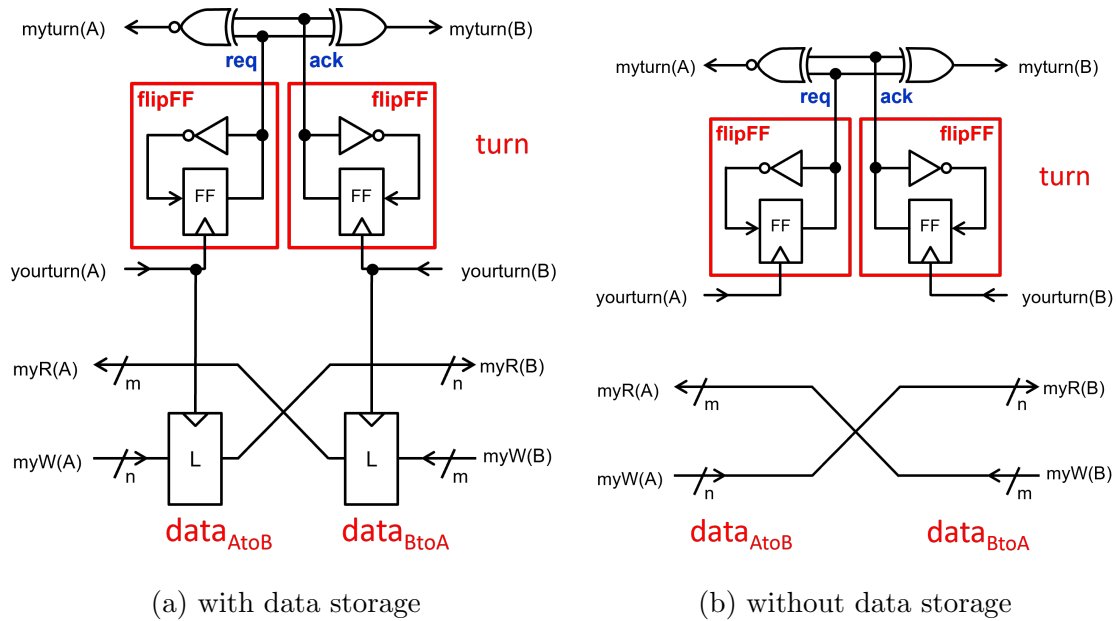


Figure 5.9: A Link with a *2-phase level-signaling bundled-data* protocol in the Click asynchronous family

Figure 5.9 shows a Link in the **Click** asynchronous family using a *2-phase level-signaling bundled data* protocol with data storage (a) and without data storage (b). We relate the circuit implementation with the semantics of Links in Section 3.1 as follows:

- The *turn* variable corresponds to two *flipFF* flipflops. At each pass (through a low to high *yourturn(p)* signal trigger), Click inverts the state of one of its *flipFF* to change internal signals, *req* or *ack*, and both its XOR and XNOR outputs, which causes *myturn(p)* to go low and *myturn(p_{peer})* to go high, giving the *turn* to p_{peer} .
- The bottom part of the figure has data latches representing Link variables $data_{AtoB}$ and $data_{BtoA}$. When *yourturn(p)* is high, it triggers the data latch

to store and transfer $myW(p)$ so that $myR(p_{peer})$ can read it. Also note that by eliminating data storage, with regards to the data storage refinement in Section 5.2.1, Figure 5.9(a) is refined to Figure 5.9(b).

Figure 5.10 shows a Link in the **Set-Reset (SR)** asynchronous family operating with a *2-phase level-signaling bundled data* protocol with data storage. The data part of the Link is the same as the Click Link. However, the logic for the *turn* variable differs. This implementation uses an SR latch. A high $yourturn(A)$ signal sets the SR latch, while a high $yourturn(B)$ signal resets the SR latch. Obviously, both signals must be high in mutual exclusion — an assumption we can specify as a timing constraint. Chapter 7 discusses our usage of timing constraints in detail. The internal *state* of the SR latch generates signals, $myturn(A)$ and $myturn(B)$.

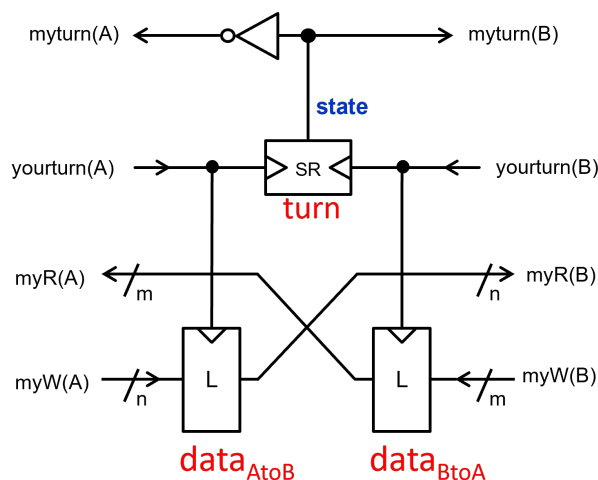


Figure 5.10: A Link with a *2-phase level-signaling bundled-data* protocol and data storage in the Set-Reset asynchronous family.

A **GasP** Link is presented in Figure 5.11. The data part is the same as Click and Set-Reset Links. The logic for the *turn* variable uses a faster SR latch split into two transistor-level circuits, *Drive-High-Keep-Low (DHKL)* and *Drive-Low-Keep-High (DLKH)*. Both circuits drive the same statewire, sw . The *DHKL* circuit on the

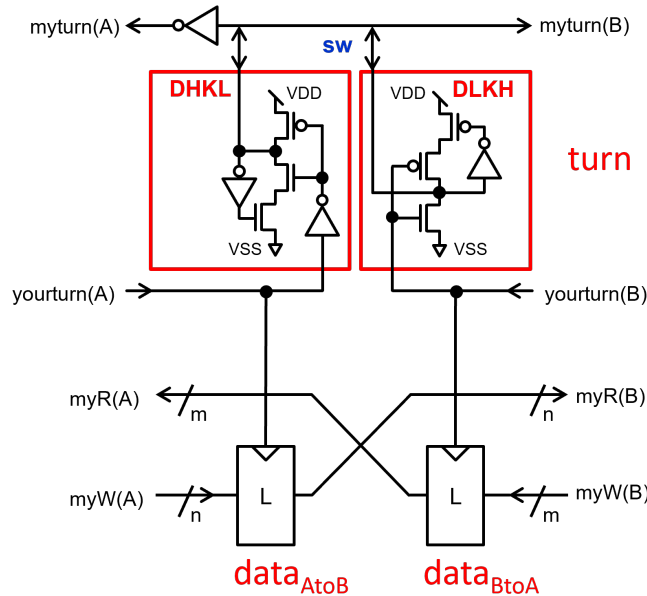


Figure 5.11: A Link with a 2-phase level-signaling bundled-data protocol and data storage in the GasP asynchronous family.

left drives sw high and keeps it low. Because it knows when it will drive sw high, it also knows when to turn the low keeper OFF to avoid a drive fight. Likewise, the $DLKH$ circuit on the right works the other way: it drives sw low and keeps it high.

Figure 5.12 shows the signal waveforms timed from left to right, with interface signal names in black and internal names in blue. The light-red vertical bars mark the activities from rising to falling $yourturn(A)$ and $yourturn(B)$ signals, which act as local latch and flipflop “clocks.” The yellow-colored boxes in the data waveforms indicate when data **must** be valid. The yellow boxes for $myW(A)$ and $myW(B)$ mark “clock” versus data setup and hold times, relevant when the Link has data storage. Note that $myR(A)$ and $myR(B)$ must be valid until the Link has stored (Figure 5.9(a)) or transferred (Figure 5.9(b)) the corresponding results — $myW(A)$ for $myR(A)$, $myW(B)$ for $myR(B)$.

Crucially, and integral to Links and Joints [60], differences in how each circuit family implements this protocol are *in-visible* at the interface: same protocol, same

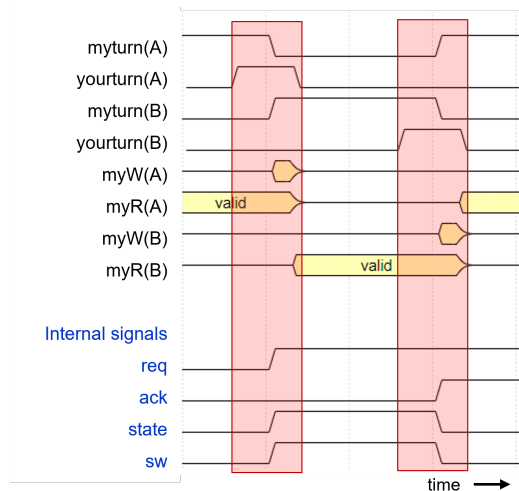


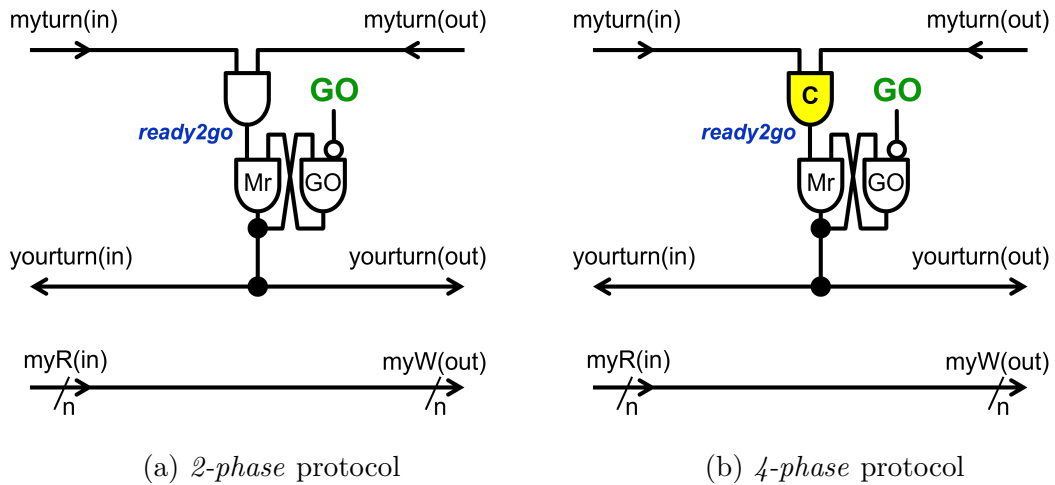
Figure 5.12: Waveforms showing the pattern of interface signals, internal signals, and data validity for a bidirectional Link with a *2-phase level-signaling bundled-data* protocol and data storage. The internal signals *req* and *ack* are for *Click*, *state* for *Set-Reset* and *sw* for the GasP asynchronous family.

interface!

5.3.2 Circuit implementations of Joint *COPY*

Joint *COPY* in Figure 3.4 is a Link-Joint library element that copies data from its port *in* to port *out* when it has the *turn* on both ports *in* and *out*, and *go*-permission. Its gate implementation can either be *2-phase* or *4-phase* based on the protocol used by the Links at its ports *in* and *out*.

Circuit implementation for Joint *COPY* in Figure 5.13(a) uses a *2-phase* communication protocol while that in Figure 5.13(b) uses a *4-phase* communication protocol. For Joint *COPY* to act, either circuit uses a *MrGO* circuit element [43, 60] to arbitrate between action denial ($\neg go$) or action permission (*ready2go*). Note that *MrGO* requires arbitration ONLY when *ready2go* is high, and *GO* is low because then it must decide whether to stop the computation by doing nothing or continue the computation by then making the *yourturns* high. Permission, if granted, persists until

Figure 5.13: Joint *COPY* gate implementation.

internal signal *ready2go* goes low.

The main difference between the *2-phase* and *4-phase* implementations is the logic that drives the Joint’s action permission, *ready2go*. In the *2-phase* version Figure 5.13(a), it takes one of the input signals — in this case, either $\neg myturn(in)$ or $\neg myturn(out)$ — to lower *ready2go* because the circuit implementation uses an *AND* gate. In the *4-phase* version Figure 5.13(b), it takes all the $\neg myturn$ signals of the ports that participate in the action — in this case, both $\neg myturn(in)$ and $\neg myturn(out)$ — to lower *ready2go* because the circuit implementation uses a *C*-element. A *C*-element is a special state-holding asynchronous circuit element, whose output *when low* remains low until all its inputs are high, and *when high*, remains high until all inputs are low [66]. Note that *C*-elements in *4-phase* Joints wait until all *turn* inputs have changed before allowing the Link ports to relinquish their *turn*.

In the datapath for Joint *COPY*, we connect *myR(in)* to *myW(out)* because the data computation is a copy function. Note that in the cases of other data computations, signal *ready2go* may be delayed to match the time for the data computation.

As is typical for *level-signaling bundled-data*, matching half the computation delay suffices because *ready2go* is used twice per action.

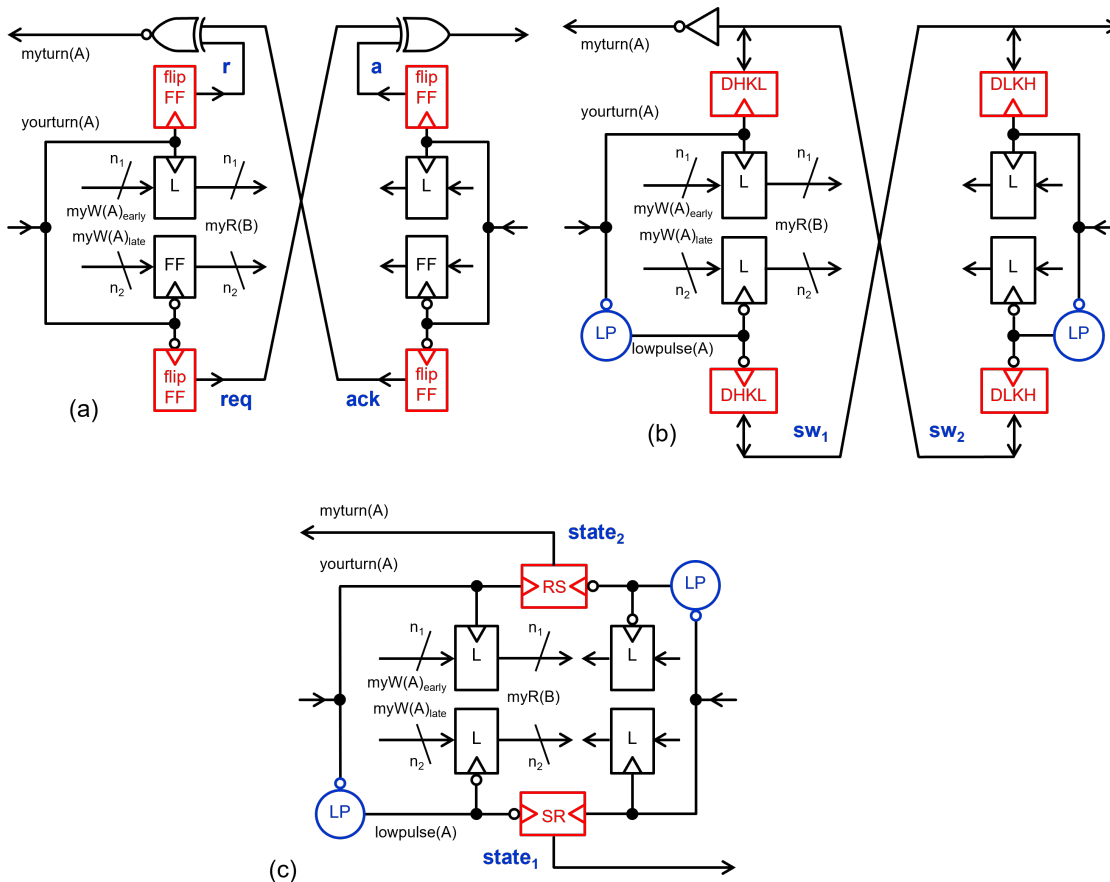


Figure 5.14: A Link with *4-phase level-signaling bundled-data* protocol and data storage in Click (a), GasP (b), and Set-Reset (c) asynchronous families.

5.3.3 4-phase level-signaling bundled data Link circuits

We present Link circuits for *4-phase level-signaling bundled data* circuit versions of the Links in Section 5.3.1. Figure 5.14 shows the *4-phase level-signaling bundled data* implementations in asynchronous families: Click (a), GasP (b), and Set-Reset (c).

The relation between guarded command specifications and circuit implementations for *4-phase* port communication with respect to the guarded command terms

defined in Section 5.2.2 are as follows:

- The terms $myturn(p)$, $myR(p)$, $myW(p)$ become circuit signals $myturn(p)$, $myR(p)$, and $myW(p)$.
- Term $midtturn(p)$ becomes $\neg myturn(p) \wedge yourturn(p)$.
- Term $halfturn(p)$ become signal transitions $yourturn(p) \uparrow$; $myturn(p) \downarrow$.
- Term $yourturn(p)$ becomes signal transitions $yourturn(p) \downarrow$; $myturn(p_{peer}) \uparrow$, where p_{peer} is A if port p connects to Link port B , and otherwise B .

The Click *4-phase level-signaling bundled data* Link in Figure 5.14(a) uses four *flipFF* gates to accomplish the *turn* variable operations in the circuit. While the equivalent GasP Link in Figure 5.14(b) uses two *DHKL* and two *DLKH* gates to accomplish the *turn* variable operations in the circuit and the Set-Reset Link in Figure 5.14(c) uses two *SR* gates to accomplish the *turn* variable operations in the circuit. Set-Reset and GasP require a low pulse generator (**LP**), which is unnecessary in Click due to the use of flipflops.

The circuits in Figure 5.14 split *4-phase* communication data into *early* and *late* data. This data split is a straightforward extension of the refinement in Section 5.2.2. Figure 5.15 depicts the data validity for a unidirectional *4-phase* Link with communication data split into *early* and *late*. The waveforms start with the *turn* at port A and relinquish the *turn* from A to B then B to A , as indicated by the two light-red vertical bars.

At best, $myW(A)_{early}$ is valid before $yourturn(A)$ goes high in the Joint connected to this Link's port A , making $myR(B)_{early}$ valid as soon as $yourturn(A)$ goes high. Worst-case, $myR(B)_{late}$ is valid towards the end of the 4-phase communication on

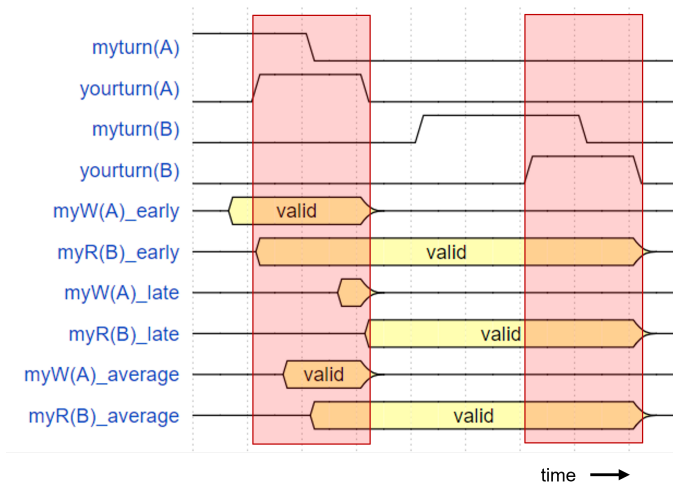


Figure 5.15: Data validity for a unidirectional 4-phase Link with communication data split into *early* and *late*. $myW(A)_{average}$ and $myR(B)_{average}$ reflect the average data validity without data splitting.

the Joint connected to this Link’s port A , that is, when $yourturn(A)$ goes low. On average, data on $myR(B)$ become valid somewhere in between.

A designer may exploit knowledge about early, average, and late validity of data in multiple ways. For instance, making data bits that the receiving Joint uses in its flow control available early on may save time for delay matching. By suppressing hazards, late data may provide a better power and energy profile. Overlapping *early* and *late* bits may support just-in-time delay matching [21].

5.4 Chapter Contributions

The research effort for our refinement process from Link-Joint networks into abstract gate-level implementations is joint work with my supervisor. These are largely mine:

- I implemented and validated the various refinement examples in Verilog.
- I specified the abstract gate implementations for the *2-phase level-signaling bundled data* Link-Joint circuits with the formalization presented in Chapter 3.

This formalization keeps in line with the circuits shown in [60].

- I implemented and validated circuit implementations for *2-phase*, *4-phase*, *level-signaling*, and *bundled-data* protocols in Verilog. We implemented more asynchronous circuit families than those presented in this chapter. *RSFQ* follows in Chapter 6 and *Micropipelines* and *Mousetrap* follow in Chapter 8.

Chapter 6: Test, Debug, and Initialization

We consider testing an activity to be carried out throughout the entire design process. In this dissertation, we are interested in connecting test and debug mechanisms at all levels of abstraction in our design flow instead of concentrating only on gate-level testing. This chapter illustrates our implementation of a uniform test approach in which we model and relate abstract test points at higher levels to test points at the gate level where standard asynchronous testing methods [25, 54–56, 64, 79] are used. We seek to connect test points at different abstraction levels to reduce test complexity in hardware. We are not just interested in testing the logical functionality of the entire design but also in guiding how testing is done on the hardware. This approach enables us to harmonize design with test and debug, starting at the program (high level) and flowing down to lower design levels, that is, to Link-Joint networks and ultimately to Link-Joint circuit implementations and chips.

As depicted in Figure 6.1, our test approach blends software interactive code debugging with integrated test and debug in the Link-Joint model [60, 61] and with DfT techniques such as *scan test* [10] used at the circuit and chip level. Our approach extends the test and debug solution built into the Link-Joint model [60]. This chapter starts by summarizing the test solution provided in the Link-Joint model in Section 6.1 and then details our extension to uniformly apply the test solution to all the different abstraction levels in our design flow. Sections 6.2 and 6.3 demonstrate how this uniform test and debug approach applies to functional as well as structural testing.

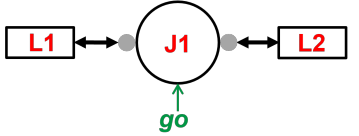
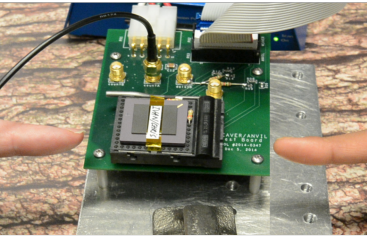
| | |
|--|--|
| <pre>defproc FIFO(){ /* channels */ chan(int<8>) b1; chan(int<8>) b2; chan(int<8>) b3; ... dataflow { b1 -> b2; b2 -> b3; b3 -> b4; ... } }</pre> | <p>To test software</p> <ul style="list-style-type: none"> so many lines – so few exports <p>use</p> <ul style="list-style-type: none"> interactive code debug to read/write states and set breakpoints for single- and multi-step tests |
|  | <p>To test Link-Joint networks</p> <ul style="list-style-type: none"> so many Links and Joints - so few external ports <p>use</p> <ul style="list-style-type: none"> Joint's go signal and Link variables to enable and disable actions and access states for single- and multi-step tests |
|  | <p>To test hardware</p> <ul style="list-style-type: none"> so many wires – so few pins <p>use</p> <ul style="list-style-type: none"> scan <ul style="list-style-type: none"> to minimize the number of pins to read/write states and go MrGO to permit and prohibit actions for single- and multi-step tests |

Figure 6.1: Our uniform test approach harmonizes program interactive code debug, Link-Joint action-state control, and circuit and chip level Design-for-Test (DfT) techniques such as scan test.

6.1 Existing State-Action Test and Debug Approach

Testing solutions often confuse state and action parts, making it difficult to separate them to minimize test access costs or refine and reuse test solutions for debugging [60]. The test and debug of asynchronous circuits is usually challenging because their states and actions are not global but distributed over space and time. This distribution makes it challenging to identify *test points*, that is, places to observe or alter state and places to pause or start system actions. Roncken et al. built a testing solution into the Link-Joint model. By ensuring a separation of states (in Links) and actions (in Joints), they “naturalized” test and debug access and tied both to the *scan* DfT method [43, 60].

At the Link-Joint level, Roncken et al. [43, 60] clarified that starting and stopping

system actions is done at the Joints and, when stopped, state variables stored by Links can be read and written. Therefore, the test points at this abstraction level include *Link variables* for initialization and observation and *go-control* to pause and start Joint actions. For example, Figure 6.2 shows a Link-Joint network for a two-stage FIFO. Through external access shown as green arrows in the figure, we observe and initialize Link variables, *turn* and *data*, in Links *L1* and *L2*. Also, through *go*-permission, shown also with a green arrow, we can control the permission of actions in the network at Joint *J1*.

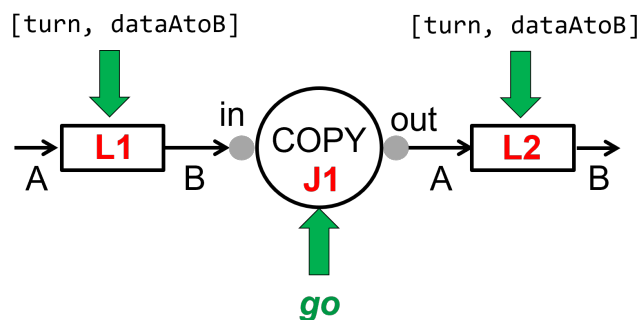


Figure 6.2: A Link-Joint network for a one-stage FIFO with one Joint, *COPY*, and two Links. The green arrows depict the external access to Link states and *go*-control on the Joint.

Roncken et al. connected this external state-action control to the gate level. The Link-Joint model provides well-defined boundaries and clear interfaces, which are also identifiable at the circuit level. The circuit uses shift registers to scan in and out state and uses a special circuit element, *MrGO*, to control the permission of actions in the circuits [60, 61]. Figure 6.3 shows a Set-Reset *2-phase bundled-data* one-stage FIFO. Scan chains shift bits in and out of the circuit serially, as Figure 6.4 illustrates. We can observe and initialize the *turn* variable by reading and writing the SR latches colored green in Figure 6.3 and the data variables by reading and writing the data latches, *L*, also colored green in the figure. We permit or prohibit the Joint's action

by writing a bit value of 1 (TRUE) or 0 (FALSE) to the Joint's *GO* signal.

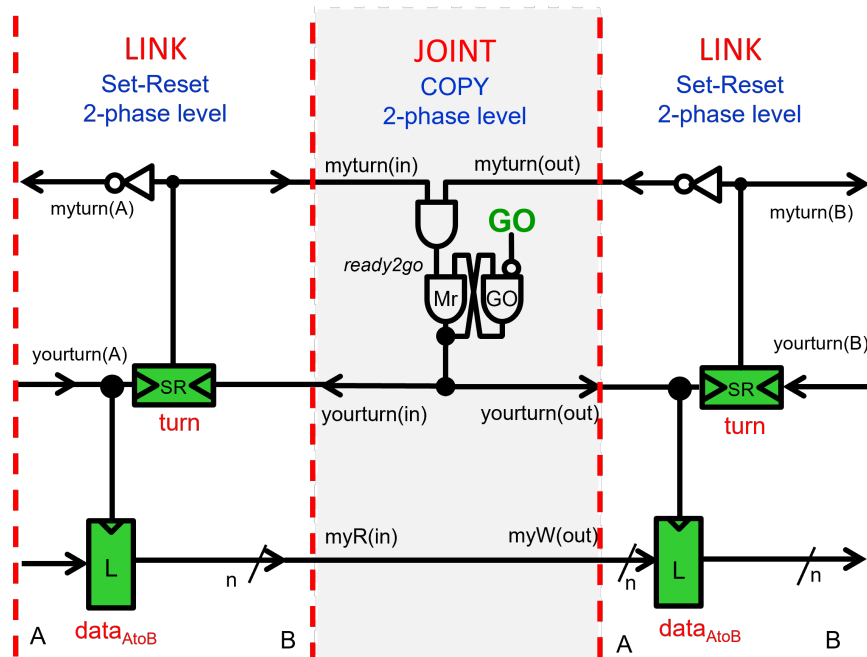


Figure 6.3: A circuit for a one-stage FIFO using a *2-phase bundled-data* protocol in the Set-Reset asynchronous circuit family. The latches colored green have *scan* access, and the *MrGO* gate decides whether to permit or prohibit the action of the Joint based on the *myturn* inputs and the external *GO* signal.

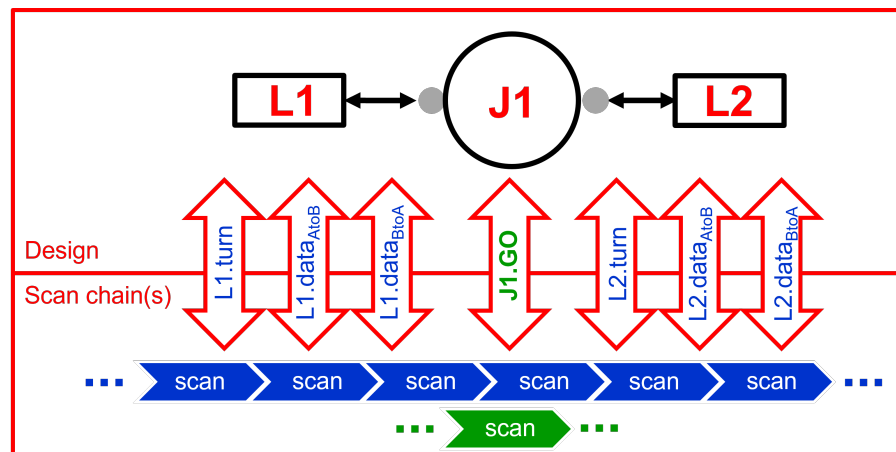


Figure 6.4: The one-to-one relation between Link variables and Joint *go*-control and circuit-level scan chains. The scan chains shift data in and out serially but may use parallel read and write from and to the Link-Joint circuit implementation.

6.1.1 Connecting the Existing Test Solution to the Program Level

Much like for any other programming activity, the ability to interactively set and observe states, as well as single- or multi-step through code, offers benefits for test and debug. Testing at the program level provides designers with quick insights into the behavior of the design. Combining this software debug approach with the Link-Joint network and circuit's test control and observation features shown in Figures 6.2 and 6.3 provides a uniform test approach for the entire design flow.

First, we identify Link-Joint corresponding state-action test access points at the program level: *variables* for initialization and observation and *breakpoints* to pause and start the program execution. Figure 6.5 gives a program, a control-flow version, for a one-stage FIFO, as seen in Figure 4.16 of Chapter 4. The *onebuf* process accepts data over input channel *L* and returns a copy of the data over output channel *R*. Variable *x* is internal to process *onebuf* and represents the data. So, initializing and observing the data requires test access to *x*. To pause and start process *onebuf* so that we can initialize and observe *x* requires breakpoints in the program for *onebuf*. We can set a breakpoint on *x*, pausing the program whenever *x* is assigned a value by either the program or the external test environment.

```

1 defproc onebuf(chan?(int) L; chan!(int) R) {
2   int x;
3   chp {
4     *[ L?x; R!x ]
5   }
6 }

```

Figure 6.5: A CHP program for a one-stage FIFO buffer from Figure 4.16.

To run our high-level ACT programs, we use a simulator called ACTSIM [33], a part of the Yale ACT toolkit. ACTSIM can simulate ACT programs with any

sublanguage supported in the ACT toolkit. It provides GNU debugger-like features, such as *watch*, *set*, *breakpt*, *step*, *advance*, and *cycle*, which are helpful for testing and debugging our source programs. With ACTSIM, we can single-step through the program, observe variables, and control their values. Note that, unlike other debuggers where breakpoints are at code locations, in ACTSIM, breakpoints are currently on variables. Therefore, the simulator stops whenever the variable used as a breakpoint is assigned.

Our approach offers uniform access to tests across different abstraction levels. We can create test sequences at a high level that can be refined while designing. Test sequences are short test bursts with local reinitialization. A test sequence typically starts with prohibiting all actions, then initializing data and setting breakpoints. The test sequence proceeds by letting the design run for some time, for a specific number of execution cycles, or until breakpoints prevent further progress. At the end of the run, the test reports the states of observed variables. The specifics of the test sequences become more detailed as we move down the design flow (refinement). We can make very specific test sequences, not just for the entire design but also for smaller portions of the design, by knowing the boundary points, specifying the breakpoints that define the design portion, and ensuring that data neither enter nor exit at these boundaries.

Using test points, initialization becomes straightforward and flexible. Utilizing our flexible initialization support, as explained in Section 3.3, we can derive any starting state by setting desired values for the program and debug variables. Flexible initialization becomes even more accessible with the introduction of program location labels, which allow jumping to arbitrary program locations, which are the program equivalent to the turn variables in the Link-Joint model. An example of the use of program labels follows in Section 6.3.

6.2 Functional Testing – Asynchronous Ring FIFO example

Functional testing in this context refers to determining the correctness of functional behavior, including performance such as throughput and energy consumption. We check the design’s performance for specific data, checkpoints, and time execution windows. In this section, we illustrate a throughput test approach by defining our test points and test sequence at the program level and translating both to the Link-Joint level and circuit level.

In this section, we use the ring FIFO design as the base for discussing the throughput test. In this example, throughput relates to how often data items pass a given location in the ring FIFO. We use a ring FIFO with a storage capacity of up to four data items. It operates by exchanging data for space. The ring FIFO can be inactive either for lack of data, when it stores zero data items, or for lack of space, when it stores four data items. It can also become inactive when we do not permit it to act. When active, the ring circulates data at full-native speed.

6.2.1 Program Level

The ring FIFO is a data-oriented design and, thus, can be specified using the *dataflow* language in ACT. As of the time of writing this dissertation, *dataflow* programs are not natively simulated in ACTSIM; instead, they have to be converted to their functionality equivalent version in CHP, which is made possible by the tool, *dflowmap*, in the ACT toolkit [33].

Figure 6.6 shows the *dataflow* program for the ring FIFO and the associated simulated CHP version. Recall from Chapter 4 that channels are not Links! They are purely for communication and synchronization and do not store data. The *func_Copy* process receives data from its input channel, *in*, and stores the data in variable *x*

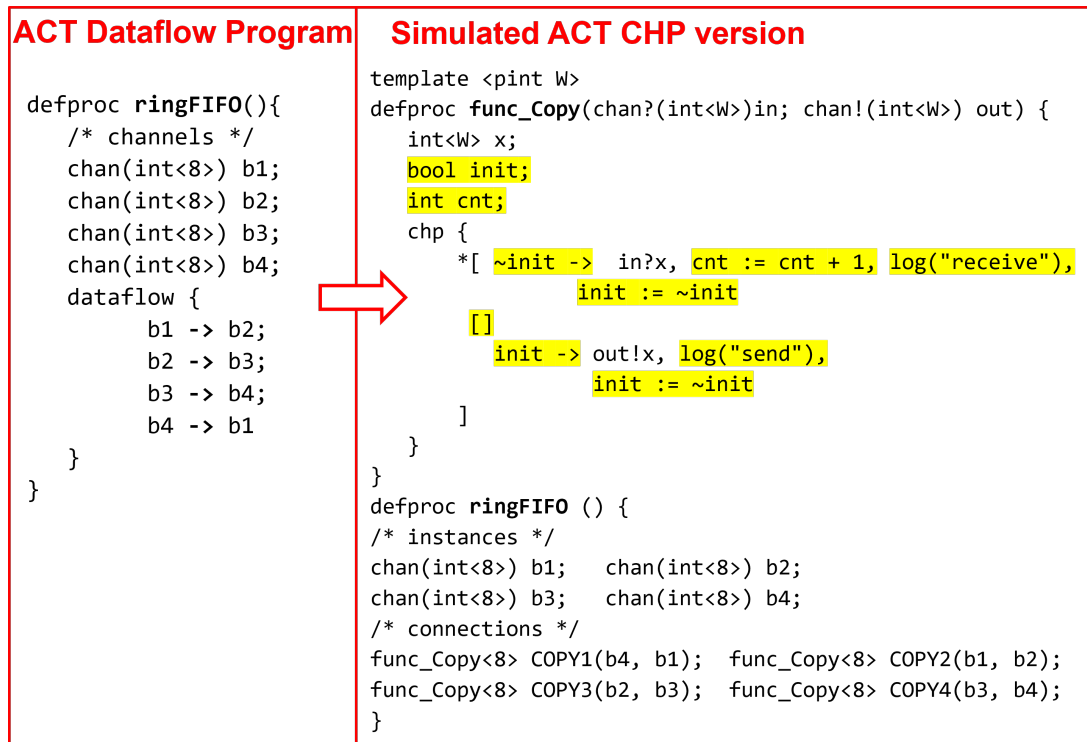


Figure 6.6: ACT program for an asynchronous ring FIFO with a storage capacity of up to four 4-bit data items. The left panel shows the ring FIFO specified as a *dataflow* program. The right-side panel shows a functionally equivalent translation by *dflowmap* [33] to which we added test augmentations (in yellow).

before sending the value of x out over the output channel, out . To aid debugging and initialization, we have augmented the CHP program with some test extensions highlighted in yellow. These include *log* statements, variable cnt to track the throughput, and variable $init$ to indicate the order of actions in the process.

The test points here are variable x , to observe and initialize data, variable cnt , to observe throughput, and variable $init$, to control the sequence of send and receive actions in each *func_Copy* process instance, $b1$ - $b4$. Using ACTSIM, we can simulate the FIFO functionality and throughput using test sequences that combine the following ingredients:

- initialize the ring with varying number of data items

- advance the simulation for some time period or execution cycles
- stop at breakpoints and observe watched variables and channels
- optionally reinitialize the variables and continue from breakpoints
- advance again or continue until there is no more progress to be made

These test points and test sequences are the basis for tests in the next abstraction level. We translate test points and test sequences to the corresponding structures at Link-Joint and circuit levels.

6.2.2 Link-Joint Level

Figure 6.7 shows the Link-Joint network for the ring FIFO and the external test access for debug and initialization. To reuse the test strategy that we developed at

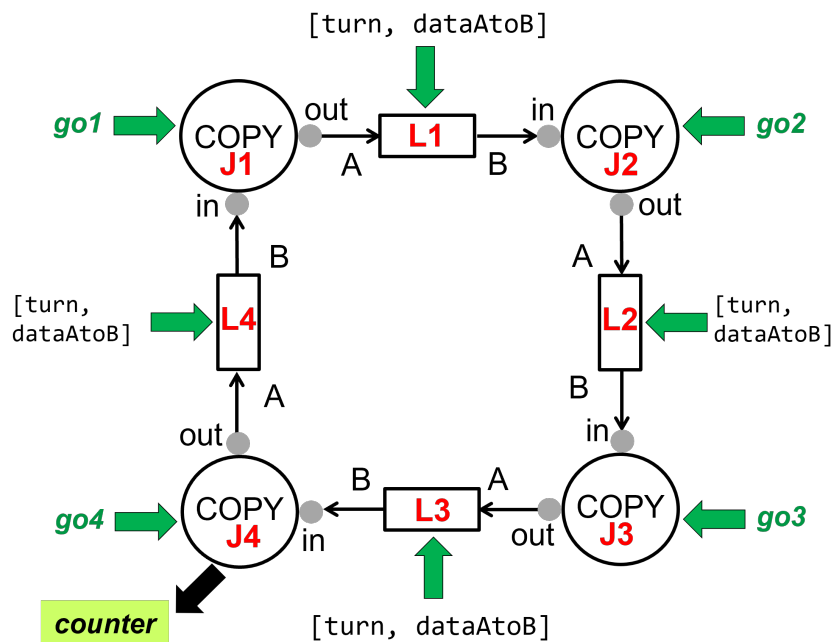


Figure 6.7: The Link-Joint network for an asynchronous ring FIFO with a storage capacity of up to four data items and its external test access.

the program level, we translate our program-level test points to Link-Joint level test points. Because of the state-action separation in the Link-Joint model, the storage from process *func_Copy* is pushed out into the Links, leaving just the copy action to the Joints.

For each data variable x in the program's *func_Copy* process instances *COPY1* through *COPY4*, we now observe and initialize data variable $data_{AtoB}$ in Links *L1* through *L4*. For debug variables *init* in *COPY1* through *COPY4*, we now initialize the *turn* variables in Links *L1* through *L4*. Simulation cycle advancement remains the same, but to start and stop the execution, we now enable and disable *go* signals, *go1* through *go4* in Joints *J1* through *J4*.

Because throughput counters are common test features, variable *cnt* is a special DfT structure responsible for its own counting logic and state information. The Joint's action triggers it, but it is not part of the Joint proper and doesn't interfere with the action and performance of the Joint [61]. Therefore, Figure 6.7 shows the counter outside Joint *J4*, which is the counter that we monitor to determine the throughput.

6.2.3 Ring FIFO Throughput Simulation

As mentioned in Section 6.1.1, we use ACTSIM for program-level simulation. Having defined our test points and test sequence, we set up the simulation to run for 1000 execution steps, after which we read the value of the throughput counter, *COPY4.cnt*. We use the same test setup at the Link-Joint level but simulate this in Verilog.

Figures 6.8 and 6.9 show the initialization and simulation of the ring FIFO as an ACTSIM script and a Verilog testbench, respectively. Both figures depict simulations with two data items in the ring and execute for 1000 execution steps.

To analyze and characterize the performance of the ring FIFO for both the pro-

```

1 watch b1 b2 b3 b4 COPY4.cnt
2 set COPY1.init 1
3 set COPY2.init 0
4 set COPY3.init 1
5 set COPY4.init 0
6 set COPY1.x0 1
7 set COPY3.x0 3
8 advance 1000
9 mget COPY1.x0 COPY2.x0 COPY3.x0 COPY4.x0 COPY4.cnt

```

Figure 6.8: Sample ACTSIM script to initialize and simulate the ring FIFO at the program level, using the CHP program in the right panel of Figure 6.6.

gram and Link-Joint network, we use *Canopy graphs* [20, 77]. A Canopy graph is a tool used in the analysis of asynchronous pipelines to model and optimize system performance. It provides a visual representation of the system’s architecture, allowing for the identification of potential bottlenecks and the assessment of slack-matching opportunities.

Figure 6.10 shows a Canopy graph relating throughput to occupancy in the ring. The throughput count in the graph is the number of data items that cross the counter during 1000 execution steps in their respective simulation environment, while the occupancy count is the number of distinct data items in the ring during the simulation. The throughput of a pipeline – a ring in this case – is affected by the total number of data items in the ring and also by the total number of stages without a data item, “spaces” [20].

Consequently, the ring FIFO’s throughput count is zero when there are zero or four valid data items in the ring. The throughput count is zero because with zero valid data items, there is nothing to exchange, and with four valid data items, the ring is FULL, and there is no space to copy the data into. The throughput of the ring with one data item (one FULL) is limited to the amount of time it takes for the single data item to move through the ring completely. Similarly, a ring with three

```

1 module testbench;
2 reg go1, go2, go3, go4;
3 wire [3:0] L1_ABin, L2_ABin, L3_ABin, L4_ABin, L1_ABout,
   L2_ABout, L3_ABout, L4_ABout;
4 initial begin
5   $dumpvars();
6   go1 = 0; go2 = 0; go3 = 0; go4 = 0;
7   #10
8   go4 = 1; go1 = 1; go2 = 1; go3 = 1;
9   #990
10  $finish();
11 end
12
13 Link L1 (.Amyturn(L1_Ame), .Ayourturn(L1_Ayou), .Bmyturn(
   L1_Bme), .Byourturn(L1_Byou), .ABin(L1_ABin), .ABout(
   L1_ABout), .init_turn(1'b1), .init_AB(4'd1));
14 Link L2 (.Amyturn(L2_Ame), .Ayourturn(L2_Ayou), .Bmyturn(
   L2_Bme), .Byourturn(L2_Byou), .ABin(L2_ABin), .ABout(
   L2_ABout), .init_turn(1'b0), .init_AB(X));
15 Link L3 (.Amyturn(L3_Ame), .Ayourturn(L3_Ayou), .Bmyturn(
   L3_Bme), .Byourturn(L3_Byou), .ABin(L3_ABin), .ABout(
   L3_ABout), .init_turn(1'b1), .init_AB(4'd3));
16 Link L4 (.Amyturn(L4_Ame), .Ayourturn(L4_Ayou), .Bmyturn(
   L4_Bme), .Byourturn(L4_Byou), .ABin(L4_ABin), .ABout(
   L4_ABout), .init_turn(1'b0), .init_AB(X));
17
18 COPY J1 (.in_myturn(L1_Bme), .in_yourturn(L1_Byou), .
   in_myR(L1_ABout), .out_myturn(L2_Ame), .out_yourturn(
   L2_Ayou), .out_myW(L2_ABin), .go(go1));
19 COPY J2 (.in_myturn(L2_Bme), .in_yourturn(L2_Byou), .
   in_myR(L2_ABout), .out_myturn(L3_Ame), .out_yourturn(
   L3_Ayou), .out_myW(L3_ABin), .go(go2));
20 COPY J3 (.in_myturn(L3_Bme), .in_yourturn(L3_Byou), .
   in_myR(L3_ABout), .out_myturn(L4_Ame), .out_yourturn(
   L4_Ayou), .out_myW(L4_ABin), .go(go3));
21 COPY J4 (.in_myturn(L4_Bme), .in_yourturn(L4_Byou), .
   in_myR(L4_ABout), .out_myturn(L1_Ame), .out_yourturn(
   L1_Ayou), .out_myW(L1_ABin), .go(go4));
22 endmodule

```

Figure 6.9: Sample Verilog testbench to initialize and simulate the ring FIFO at Link-Joint level, using the Link-Joint network in Figure 6.7. The Links and Joints in this testbench are instances of the Verilog modules presented in Figures 3.15 and 3.16.



Figure 6.10: Canopy graph showing the simulated throughput of the ring FIFO with varying numbers of data items at different abstraction levels.

data items (one EMPTY) shows the same throughput behavior. Though there are more items in the ring, the throughput is limited to the time that “space” moves through the ring so the data items can progress forward. In the case where the ring FIFO is half-FULL or half-EMPTY, the FIFO has optimal throughput and is twice the frequency of the quarter-FULL or quarter-EMPTY ring FIFO.

The actual values for the throughput counts for either program or Link-Joint networks do not quite matter. What is essential is that the throughput pattern stays the same regardless of the abstraction level. The Link-Joint network may appear slower than the program because it includes static delay injected for the Verilog simulation. What is important to note is that both canopy graphs look like canopies or tents and show positive throughput for all ring occupancies between EMPTY (0) and FULL (4). Generally, outside of this example, by using canopy graphs, engineers can analyze and optimize the performance of asynchronous systems more efficiently, which is crucial for improving the overall speed and reliability of computing systems.

6.2.4 Circuit Level

The ring FIFO can be and has been implemented in multiple asynchronous families [57, 62]. This section implements our ring FIFO example in a superconducting technology using the RSFQ asynchronous circuit family. Figure 6.11 shows a *2-phase bundled-data pulse-logic* RSFQ implementation of a Link and Joint *COPY*. This section summarizes the detailed description of our Link-Joint RSFQ implementation given in [57].

RSFQ Links use D2 latches to store their data. During normal operation for *DATA_AtoB*, data come into the D2 latch through *A[data]* and leave through *B[data]*. However, for initialization, test, and debug, we redirect data for D2 to come from or go to a scan chain *scan[din, shift, dout]*. The RSFQ Link implementation uses two instances of a special RSFQ gate, a nondestructive readout (NDRO) called *STATE*, for the *turn* variable. The two instances are called *TURNA* and *TURNB*. Our *MrGO*

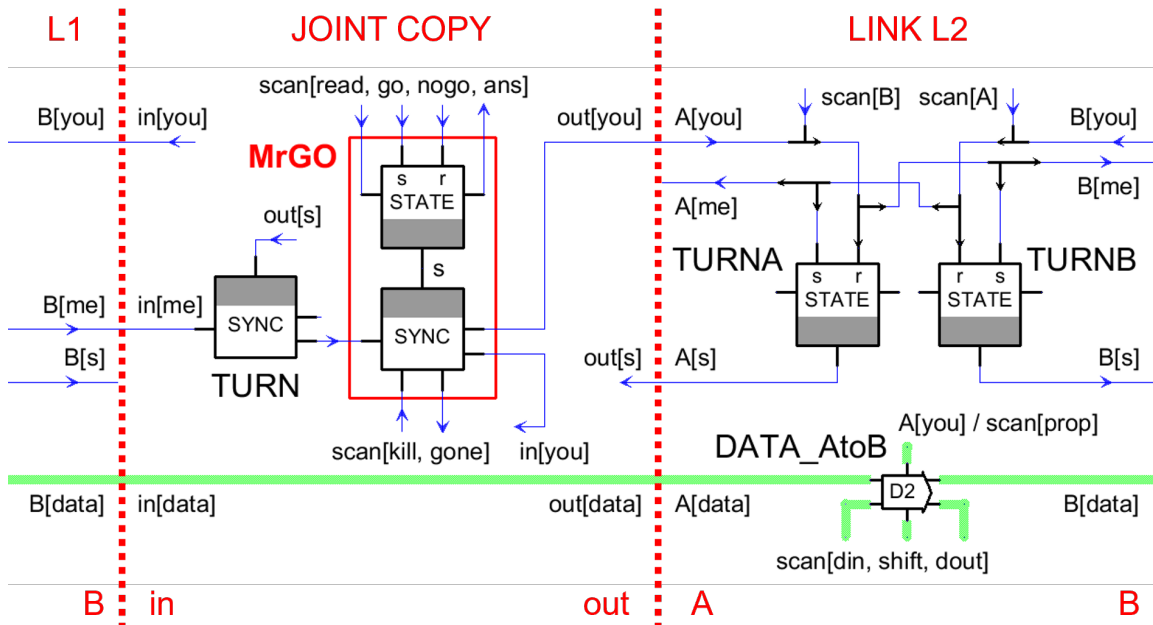


Figure 6.11: A *2-phase bundled-data pulse-logic* RSFQ (superconducting family) implementation of a Link and Joint *COPY*.

implementation in RSFQ also uses *STATE* and combines it with a special RSFQ gate called *SYNC*. The guarded command specification of *STATE* and *SYNC* follow next. Each term in these guarded commands is a Boolean, and it evaluates to TRUE when the corresponding signal carries a pulse and to FALSE when there is no pulse.

STATE (*set_state*, *reset_state*, *read_state*, *ans_state*, s_1)

set_state $\rightarrow s_1 := true ; set_state := false$

reset_state $\rightarrow s_1 := false ; reset_state := false$

read_state $\rightarrow ans_state := s_1 ; read_state := false$

SYNC (*R*, *killR*, *goneR*, *ansA*, *ansB*, s_1)

s_2 : bool

R $\rightarrow s_2 := true ; R := false$

killR $\rightarrow goneR := s_2 ; s_2 := false ; killR := false$

$s_1 \wedge s_2 \rightarrow ansA := true ; ansB := true ; s_2 := false$

In line with our test approach, we translate the Link states we have selected to observe and initialize to the specific gates for which we need scan access. We also scan in *go-nogo* signals to pause or start the circuit using *MrGO*. In addition, because test circuitry is built with the design when using the *scan* DFT test technique, scan access can be expensive in area and delay. We do not need to access all the states for useful tests. For instance, we can avoid scanning all but Link *L1*'s data by providing scan access for both the *turn* variable and the *data* variables in *L1*, while for *L2*, *L3*, and *L4*, we have scan access only for the *turn* variable. Data items will be shifted into the other Links from *L1*. This reduced scan access approach works independently of the protocol and circuit family [57, 61].

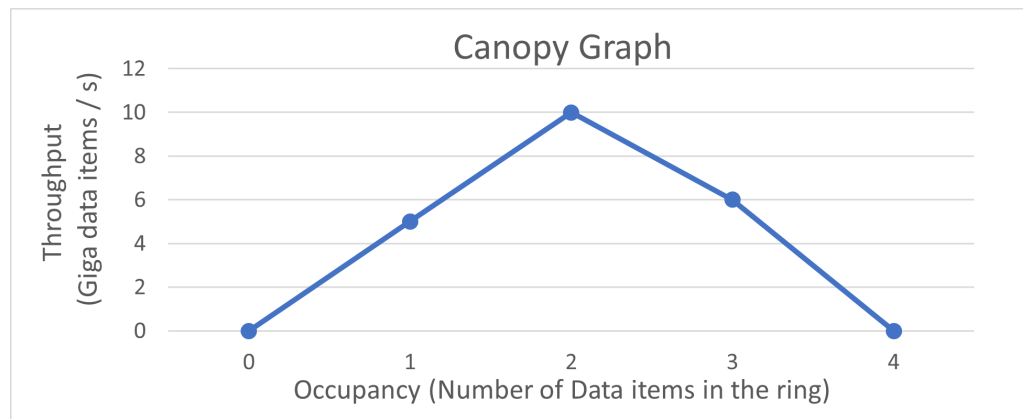
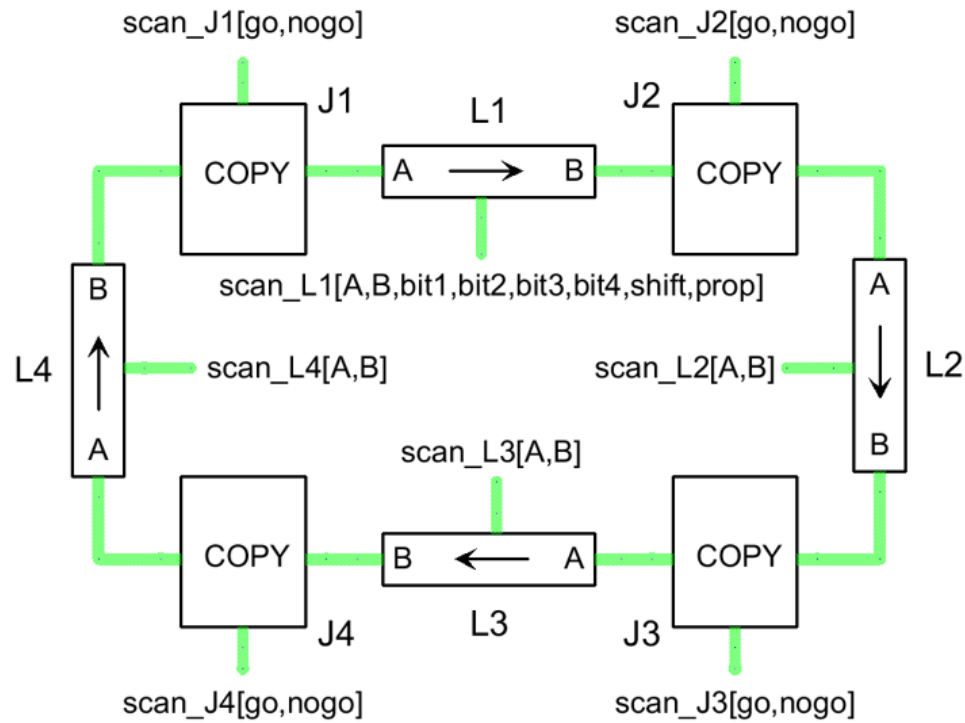


Figure 6.12: RSFQ ring FIFO implementation with reduced scan access (top) and an associated canopy graph showing the simulated throughput (bottom).

6.3 Structural Testing

Apart from end-to-end functional tests on the design, it helps to have guarantees on the internal structure and behavior of designs. Most faults models for asynchronous circuits — stuck-at faults and delay faults — are addressed by examining specific

structures in the circuit [64, 66, 79]. One way of streamlining the generation of tests for internal structures is to start at the high level by defining test sequences for basic statements in high-level programs, much like unit testing in the software world [19]. We acknowledge we do not have all the information at the high level to identify potential issues such as glitches, metastability, and timing violations. However, using the structure of the program (program statements), we can specify test requirements and expected behavior for each statement. Test details such as write values, expected read values, and the number of tests per structure can be refined at the lower abstraction levels.

To illustrate, we use a basic Greatest Common Divisor (GCD) example based on the Euclidean Algorithm, as given in Figure 6.13. We augmented the program with *labels*, *pc1* through *pc5*, to support the use of *goto* commands that enable us to jump to a labeled program location during debug. We also augmented the program with *log* statements to print information to the simulation console.

Figure 6.14 shows the Link-Joint network generated for the program in Figure 6.13

```

1  defproc gcd2(chan?(int) X,  Y; chan!(int) O) {
2    int x, y;
3    chp {
4      *[ pc1: X?x,  Y?y;
5         pc2: *[ y > x -> log("Guard 1 chosen");
6             pc3: y := y - x
7             [] x > y -> log("Guard 2 chosen");
8             pc4: x := x - y
9             ];
10         log("Out of loop");
11         pc5: O!x
12     ]
13 }
14 }
```

Figure 6.13: A CHP program for the Greatest Common Divisor (GCD) algorithm, augmented with labels and *log* statements for use in debug and test generation.

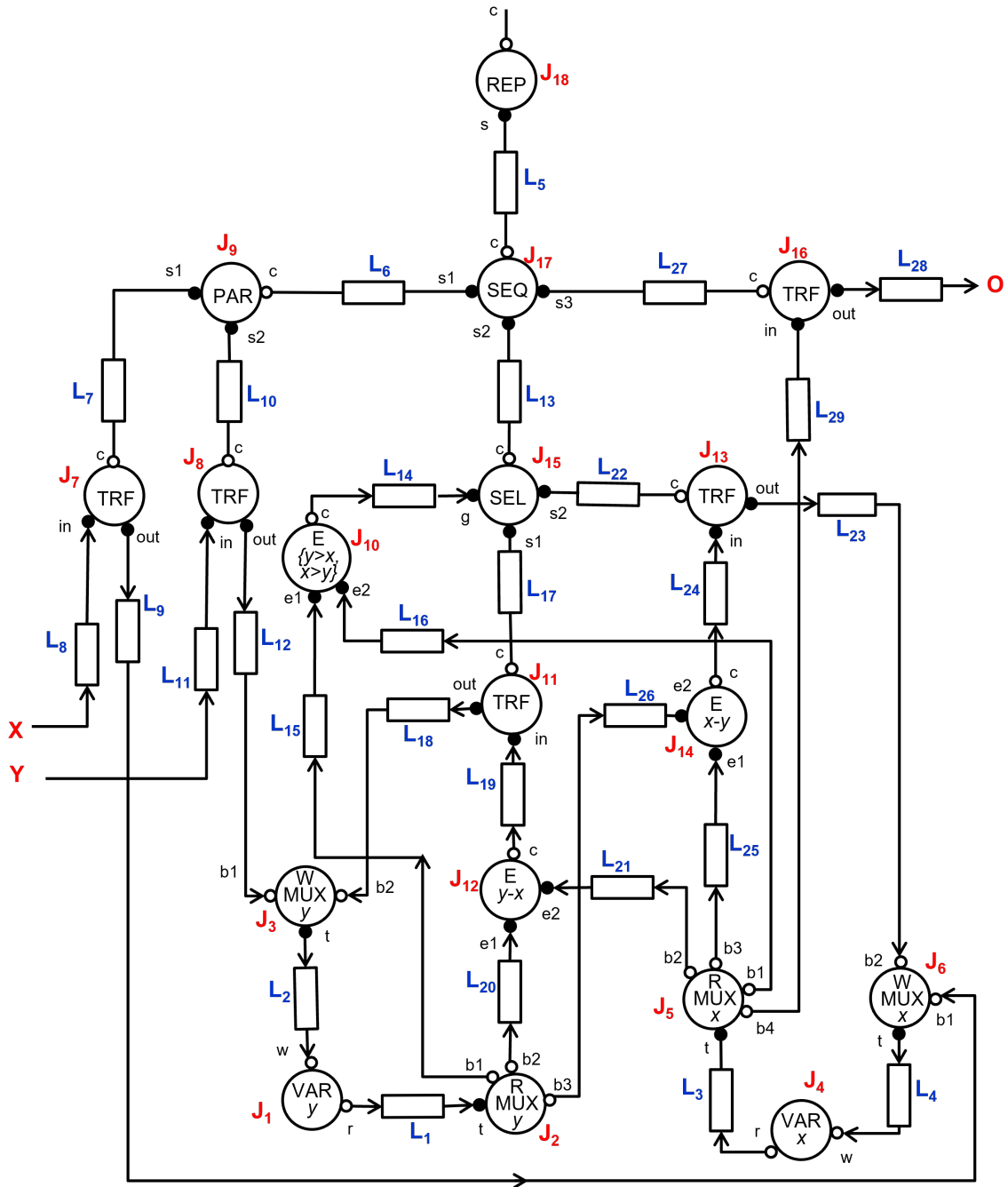


Figure 6.14: A Link-Joint network for the Greatest Common Divisor (GCD) algorithm, compiled from the CHP program in Figure 6.13, using the compilation approach explained in Chapter 4.

excluding the program augmentations added. Note that though either Link ports A or B can be *active* or *passive* as discussed in Section 3.3, in Figure 6.14, we set Link port A as the *active* port (\bullet) and Link port B as the *passive* port (\circ) for the test discussions in the following subsections. The following subsections discuss some basic statements in the program with their test requirements and test sequences. We do not discuss all the constructs in the program; instead, through a few statements, we demonstrate the test strategy.

6.3.1 Assignment

An assignment is a data operation. Therefore, we must ensure that any variable used in its expression can be read, the evaluation of the expression works, and the value evaluated is stored in the assigned variable. The GCD program, in Figure 6.13, has two assignments, $x := x - y$ and $y := y - x$. The program-level test sequence defined as the ACTSIM script in the left panel of Figure 6.15 is used to test assignments. The script gives the output in the right panel.

The script starts with watching changes on variables x and y and indicates the variables on which to have breakpoints. ACTSIM has a command, *goto*, that allows the simulation to jump to a statement at a particular label. At the command *goto pc3*, the simulator jumps to the location at program label, *pc3*. After being set as the script indicates, the simulator outputs the new values of x and y . The *cycle* command instructs the simulator to advance the execution until no progress can be made. Therefore, the simulation continues by executing $y := y - x$, displaying the new value of y , and then halting because of the breakpoint on y . The *mget x y* command instructs the simulator to display the current values of x and y . The script continues with the test sequence for $x := x - y$.

| ACTSIM Script | ACTSIM Output |
|---------------|----------------------------|
| 1. watch x y | [0] <[env]> y := 25 (0x19) |
| 2. breakpt x | [0] <[env]> x := 7 (0x7) |
| 3. breakpt y | [0] <> y := 18 (0x12) |
| 4. goto pc3 | [0] <> *** breakpoint y |
| 5. set y 25 | x: 7 (0x7) |
| 6. set x 7 | y: 18 (0x12) |
| 7. cycle | [0] <[env]> y := 7 (0x7) |
| 8. mget x y | [0] <[env]> x := 25 (0x19) |
| 9. goto pc4 | [0] <> x := 18 (0x12) |
| 10. set y 7 | [0] <> *** breakpoint x |
| 11. set x 25 | x: 18 (0x12) |
| 12. cycle | y: 7 (0x7) |
| 13. mget x y | |

Figure 6.15: Program Assignment test sequence as an ACTSIM script and its output.

The Verilog testbench in Figure 6.16 is refined from the program test in Figure 6.15, and it implements the same test sequence using the Link-Joint structures in Figure 6.14 corresponding to the program test points. The names J_n (where n is a number) correspond to the numbered Joints in the Link-Joint network. The test starts with making all the *go*-signals $low(0)$ to disable all Joint actions, then proceeds with terms corresponding to the program testbench.

The statement *goto pc3* corresponds in the Verilog testbench to giving the *turn* to the start-up port of *TRF* Joint J_{11} , which is connected to Link port B of Link L_{17} , and which triggers the expression evaluation, $y - x$, and assignment, $y := y - x$. The *set y 25* command corresponds to initializing the data value of *Var y*, which requires setting the internal Link in *Var y* to 25, ($J_{11}.st_ABin = 25$). #10 and #200 are static numbers of cycles the Verilog simulator should take before executing the following command. The injection of these cycles in our testbench ensures the network has made all the progress it needs to make before a command from the simulation is executed. Figure 6.17 shows the Verilog waveform for the assignment Verilog testbench.

```

1. initial begin
2. $dumpvars();
3. // ASSIGNMENT 1: y:= y-x
4.   go[18:1] = {18{1'b0}}; // Disable all Joint actions.
5. #10
6. // Goto pc3, which is at TRF J11 at Link port L17B, by giving the turn of L17
   to port B
7.   L17.reg_Amyturn = 0;
8.   L17.reg_Bmyturn = 1;
9.   // Set the internal Link data for y and x in VAR Joints J1 and J4
10.  J1.st_ABin = 25; // VAR y
11.  J4.st_ABin = 7; // VAR x
12.  // Enable the Joints in the runway of the assignment
13.  go[12:11] = 2'b11; // TRF J11, E J12
14.  go[5:1] = {5{1'b1}}; //RMUX y J2, VAR y J1, WMUX y J3, RMUX x J5, VAR x J4
15. #220 // Run for long enough to complete the assignment
16. // Check that VAR y J1.st_ABin has the right value.
17.
18. // ASSIGNMENT 2: x:= x-y
19.   go[18:1] = {18{1'b0}}; // Disable all Joint actions.
20. #10
21. // Goto pc4, which is at TRF J13 at Link port L22B, by giving the turn of
   L22 to port B.
22.   L22.reg_Amyturn = 0;
23.   L22.reg_Bmyturn = 1;
24.   // Set the internal Link data for y and x in VAR Joints J1 and J4
25.   J1.st_ABin = 7; // Var y
26.   J4.st_ABin = 25; // Var x
27.   // Enable the Joints in the runway of the assignment
28.   go[14:13] = 2'b11; // TRF J13, E J14
29.   go[6:4]   = 3'b111; // WMUX x J6, RMUX x J5, VAR x J4
30.   go[2:1]   = 2'b11; // RMUX y J2, VAR y J1
31. #220 // Run for long enough to complete the assignment
32. // Check that VAR x J4.st_ABin has the right value.
33. $finish();
34. end

```

Figure 6.16: Link-Joint Assignment test sequence as a Verilog testbench.

We can plug in different values of x and y to ensure the correct computation behavior. The purpose of showing the ACTSIM script and the associated Verilog testbench is to understand how the test sequence gets refined as the abstract level changes. Starting at the high level makes test and debug easier by knowing where to

plug in the standard test pattern generation methods at the circuit level and having a skeletal test sequence that can be expanded to the required level of detail.

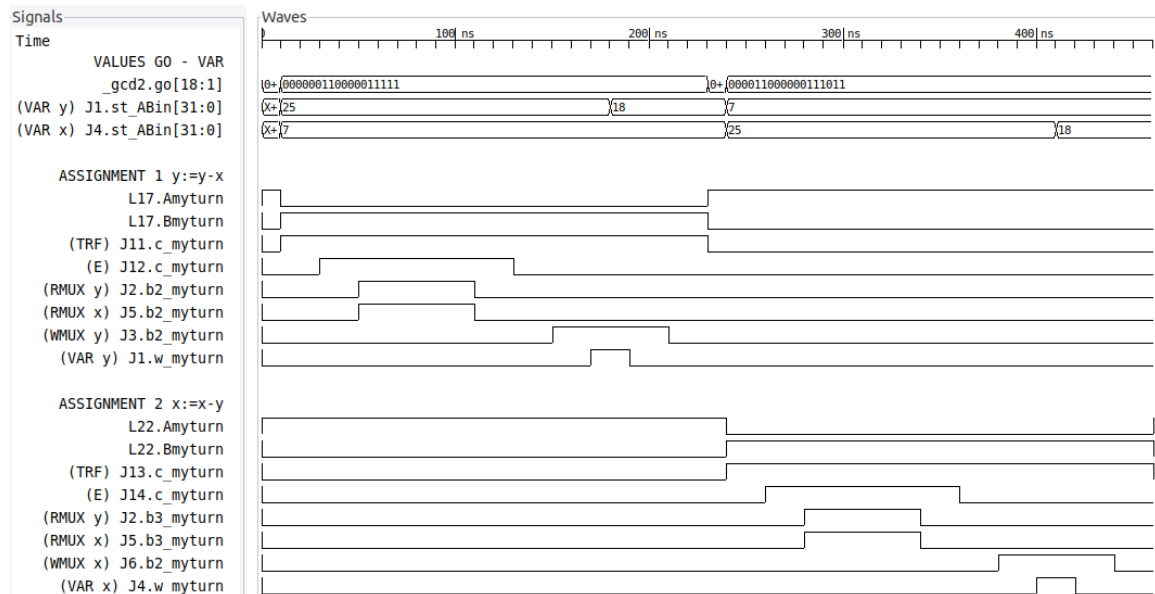


Figure 6.17: Waveforms from the simulation of the Assignment Verilog testbench.

6.3.2 Loops and Selection

Loops and selection tests require that all possible cases are reachable and that if none of the cases are chosen, execution exits the statement (for a loop) or deadlocks (for a select without an *else* statement), as required by the behavior of SELECT and LOOP in CHP (Section 4.1.1). This GCD program has a loop with two cases, from line 5 to line 9 in Figure 6.13. One case condition is $y > x$, and the other is $x > y$. If neither of these conditions holds, we exit the loop. Otherwise, we loop and test the condition again.

Figure 6.18 shows the GCD program's test sequence in ACTSIM (left) and its output (right) for the select-loop. ACTSIM command *goto pc2* gets the simulator to jump to the start of the loop. From there, we use different values of x and y to

| ACTSIM Script | ACTSIM Output |
|---------------|-----------------------------|
| 1. watch x y | [0] <[env]> y := 25 (0x19) |
| 2. breakpt x | [0] <[env]> x := 7 (0x7) |
| 3. breakpt y | [0] <> Guard 1 chosen |
| 4. goto pc2 | [10] <> y := 18 (0x12) |
| 5. set y 25 | [10] <> *** breakpoint y |
| 6. set x 7 | [10] <[env]> y := 7 (0x7) |
| 7. cycle | [10] <[env]> x := 25 (0x19) |
| 8. goto pc2 | [10] <> Guard 2 chosen |
| 9. set y 7 | [20] <> x := 18 (0x12) |
| 10. set x 25 | [20] <> *** breakpoint x |
| 11. cycle | [30] <> Guard 2 chosen |
| 12. cycle | [40] <> x := 11 (0xB) |
| 13. goto pc2 | [40] <> *** breakpoint x |
| 14. set y 7 | [40] <[env]> x := 7 (0x7) |
| 15. set x 7 | [40] <> Out of loop |
| 16. cycle | |

Figure 6.18: Program Select-Loop test sequence as an ACTSIM script and its output.

test each case. Following the test requirements, we make sure we can reach either $y > x$ or $x > y$ case statement, loop through the statement after a case is executed, and fall out of the loop. Figure 6.19 shows the Link-Joint Verilog testbench following the same test sequence as the program test. Note that in the Link-Joint network test, it is sufficient to show control reaches the start-up Link of the case statement to be executed without actually executing it. This is possible using *go* signals in the Link-Joint network. Figure 6.20 shows the waveform from simulating the testbench.


```

1.  initial begin
2.  $dumpvars();
3.  // Guard 1: y > x
4.  go[18:1] = {18{1'b0}}; // Disable all Joint actions.
5.  #10
6.  // Goto pc2, which is at SEL J15 at Link port L13B, by giving the turn of L13
   to port B.
7.  L13.reg_Amyturn = 0;
8.  L13.reg_Bmyturn = 1;
9.  // Set the internal Link data for VAR y & VAR x so that guard 1 becomes true
10. J1.st_ABin = 25; // Var y
11. J4.st_ABin = 7; // Var x
12. // Enable the Joints in the runway of the guard selection
13. go[15] = 1'b1; // SEL J15
14. go[10] = 1'b1; // E J10
15. go[5:4] = 2'b11; // RMUX x J5, VAR x J4
16. go[2:1] = 2'b11; // RMUX y J2, VAR y J1
17. #220 // Run for long enough to complete the guard selection
18. // Check that SEL gave its turn to command 1 at Link port L17A, i.e., that
   the turn for Link L17 is at Link port B.

19. // GUARD 2: x > y
20. go[18:1] = {18{1'b0}}; // Disable all Joint actions.
21. #10
22. // Goto pc2, which is at SEL J15 at Link port L13B, by giving the turn of L13
   to port B, and by restoring the turn of L17 from port A at command 1 at L17B
   to Joint SEL at port L17A.
23. L13.reg_Amyturn = 0;
24. L13.reg_Bmyturn = 1;
25. L17.reg_Amyturn = 1;
26. L17.reg_Bmyturn = 0;
27. // Set the internal Link data for y and x in VAR Joints J1 and J4
28. J1.st_ABin = 7; // Var y
29. J4.st_ABin = 25; // Var x
30. // Enable the Joints in the runway of the guard selection
31. go[15] = 1'b1; // SEL J15
32. go[10] = 1'b1; // E J10
33. go[5:4] = 2'b11; // RMUX x J5, VAR x J4
34. go[2:1] = 2'b11; // RMUX y J2, VAR y J1
35. #220 // Run for long enough to complete the guard selection
36. // Check that SEL gave its turn to command 2 at Link port L22A, i.e., that
   the turn for Link L22 is at Link port B.
37. // EXIT:
38. go[18:1] = {18{1'b0}}; // Disable all Joint actions.
39. #10
40. // Goto pc2, which is at SEL J15 at Link port L13B, by giving the turn of L13
   to port B, and by restoring the turn of L22 from port A at command 2 at L22B

```

```

to Joint SEL at port L22A.
41. L13.reg_Amyturn = 0;
42. L13.reg_Bmyturn = 1;
43. L22.reg_Amyturn = 1;
44. L22.reg_Bmyturn = 0;
45. // Set the internal Link data for y and x in VAR Joints J1 and J4
46. J1.st_ABin = 7; // Var y
47. J4.st_ABin = 7; // Var x
48. // Enable the Joints in the runway of the guard selection
49. go[15] = 1'b1; // SEL J15
50. go[10] = 1'b1; // E J10
51. go[5:4] = 2'b11; // RMUX x J5, VAR x J4
52. go[2:1] = 2'b11; // RMUX y J2, VAR y J1
53. #220 // Run for long enough to complete the guard selection
54. // Check that SEL exited by giving its turn back to its startup Link at port
L13B, i.e., that the turn for Link L13 is at Link port A.
55. $finish();
56. end

```

Figure 6.19: Link-Joint Select-Loop test sequence as a Verilog testbench.

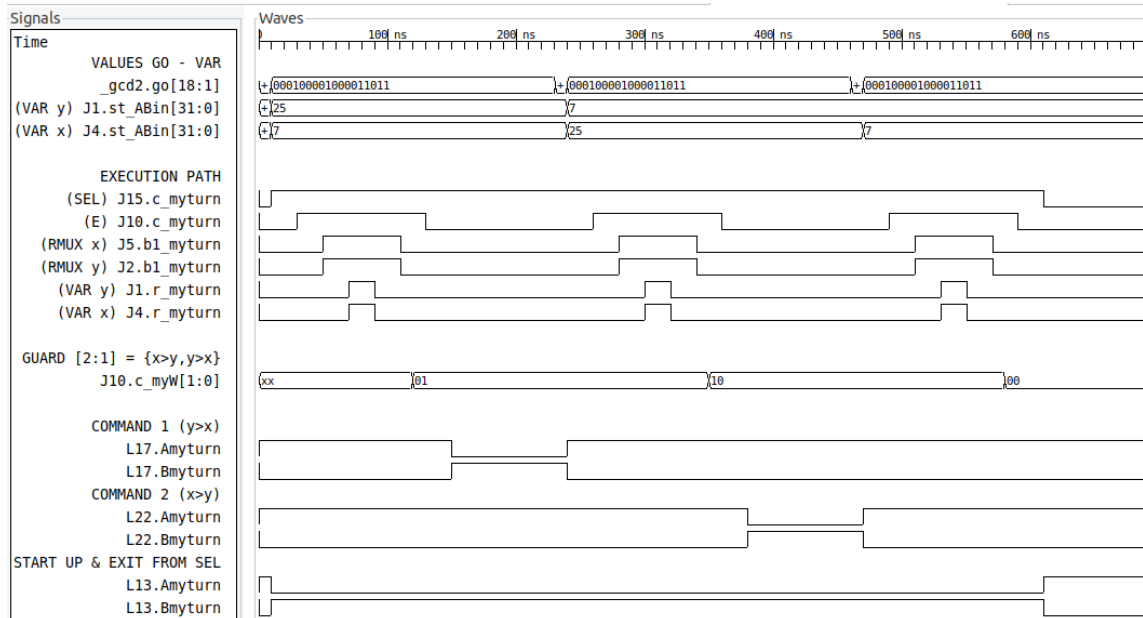


Figure 6.20: Waveforms from the simulation of the Select-Loop Verilog testbench.

6.3.3 Communication

Communication statements are basically data transfers and can be tested like assignments. This section considers their effect in a program with multiple communicating processes. When testing a single communication process in isolation, the process deadlocks when it tries to communicate because it lacks a communication peer. Deadlocking is inevitable if other processes communicating with the process under test are not simultaneously executed. So, what can we do to test communication commands in isolation?

At the Link-Joint and circuit level, we can directly unblock the communication by setting the *turn* variable or corresponding Link state variable in the circuit externally. However, that cannot be easily realized in the program simulation. Therefore, we worked with Rajit Manohar at Yale to create a new ACTSIM command, *skip-comm*, to unblock a blocked channel that is waiting for a communication partner. We unblock by skipping the blocked channel communication.

Figure 6.21 shows a test of output channel *O* of the GCD process (in Figure 6.13). The *goto pc5* command jumps the simulator just before the start of the output communication. At this point, the state of the channel is *idle* because it has not yet started the communication. We now set the value for *x*. With a *cycle* command,

| ACTSIM Script | ACTSIM Output |
|----------------|--|
| 1. watch x 0 | |
| 2. goto pc5 | [0] <[env]> x := 7 (0x7) |
| 3. set x 7 | O: idle |
| 4. get 0 | [0] <> 0 : send-blocked value: 7 (0x7) |
| 5. cycle | O: waiting sender |
| 6. get 0 | O: idle |
| 7. skip-comm 0 | |
| 8. get 0 | |

Figure 6.21: Program test sequence to test output communication *O!x* in isolation with ACTSIM.

the communication commences and gets blocked while waiting to send the value of x to another process or environment communicating with it. Because no other process communicates with process *gcd2* in this test, the process deadlocks, and no other progress can be made. However, we can see the value O wants to send and validate that it is the correct value of x , 7. With the *skip-comm O* command, we unblock the blocked channel O , and a new execution cycle can now begin. The test in Figure 6.21 allows us to test all parts in the output communication $O!x$ that are in the *gcd2* process. All such parts can be tested without any need for collaboration with a communication partner. Figure 6.22 shows the associated Verilog testbench while Figure 6.23 shows the waveforms from simulating the testbench.

```

1. initial begin
2. $dumpvars();
3. // OUTPUT COMMUNICATION:
4.   go[18:1] = {18{1'b0}}; // Disable all Joint actions.
5. #10
6. // Goto pc5, which is at TRF J18 at Link port L17B, by giving the turn of L27
   to port B
7.   L27.reg_Amyturn = 0;   L27.reg_Bmyturn = 1;
8.   // Set the internal Link data for x in VAR Joint J4
9.   J4.st_ABin = 7;
10.  // Enable the Joints in the runway of the O!x communication
11.  go[16] = 1'b1; // TRF J16
12.  go[5:4] = 2'b11; // RMUX x J5, VAR x J4
13. #100 // Run for long enough to get the value of x to output O
14.   // Check that TRF transferred both turn and data from its output port L28A
   to port L28B at output O.
15.
16. // SKIP COMMUNICATION AND RESTORE INITIAL DESIGN STATE:
17.   go[18:1] = {18{1'b0}}; // Disable all Joint actions.
18. #10
19. // Give the O Link turn back to L28A.
20.   L28.reg_Amyturn = 1;   L28.reg_Bmyturn = 0;
21. // Enable the Joints in the restoration runway
22.   go[16] = 1'b1; // TRF J16
23. #50 // Run for long enough to allow TRF J16 to give its turn back for its
   startup Link, L28, i.e., to give the turn from L28B to L28A.
24. // Check that the turn for Link L28 is at port L28A.
25. $finish();
26. end

```

Figure 6.22: Link-Joint test sequence to test output communication $O!x$ in isolation as a Verilog testbench.

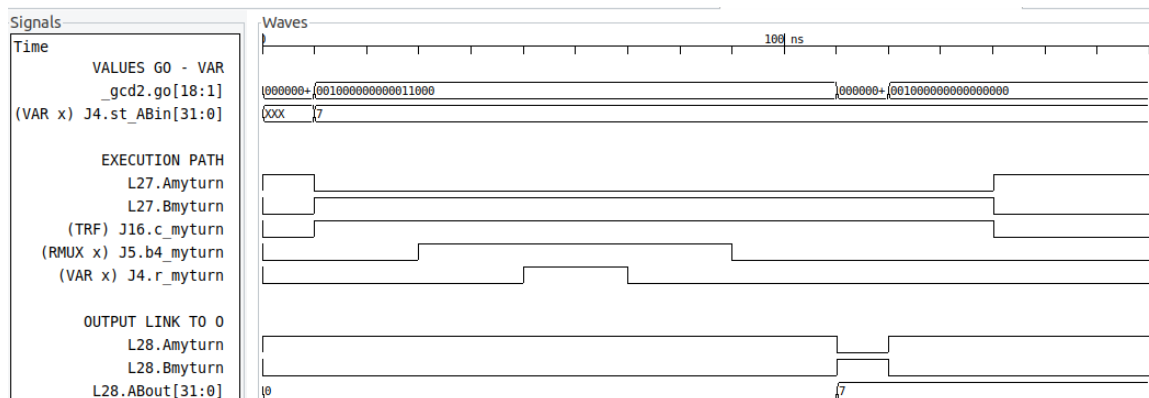


Figure 6.23: Waveforms from the simulation of the Verilog testbench in Figure 6.22

6.4 Connecting Our Uniform Test Approach to Hardware Test Methods

Our motivation for the uniform test approach from programs to circuits, illustrated in Sections 6.2 and 6.3, is to guide how testing is done in hardware. Regardless of the abstraction level, the uniformity in the test approach ensures that some key ideas remain the same and can be handed over to manage test complexity:

- start points:(1) where does the operation start, (2) what are the (key) initial values
- stop points:(1) where does the operation stop, (2) what are the (key) end values

These test points can be tied to the test pattern generation techniques [25, 54–56, 64, 79] based on hardware fault models to have targeted tests, which can be detailed later for hardware fault coverage.

6.5 Chapter Contributions

This test approach started before this dissertation [43, 60, 61]. However, its extension for all abstraction levels and the validation and simulation presented in this chapter

are a joint effort of my supervisor and me. The following contributions are largely mine.

- I extended the test approach to high-level programs, connecting test points at Link-Joint and circuit levels to the program level.
- I implemented, simulated, and validated test examples at all abstraction levels.
- While exploring test and debug using ACTSIM, I contributed new commands to the simulator. These commands include:
 1. *skip-comm*:- unblocks a channel waiting for its partner to communicate by skipping the communication. Because of this command, we can test a process in isolation.
 2. *gc-retry*:- retries guards in a deadlocked process after the external test sequence changes variables. This command allows the program to recompute its state based on the changes provided.
 3. *goto*:- jumps to a specified label for a single-threaded state. This command makes flexible initialization more accessible and simpler at the program level because the program locations are equivalent to *turn* variables in the Link-Joint model. The alternative is to augment the program with debug variables that represent locations, as we did for the example in Figure 6.6 with the introduction of debug variable *init*.

Chapter 7: From Link-Joint Circuits to ACT

The preceding chapters have detailed the different pieces of our design flow, *Qnà*, and we have demonstrated the generality and uniformity of our approach, going from high-level programs in ACT through Link-Joint networks to multiple protocols and circuit families. This chapter shows how Link-Joint circuit implementations can be brought back into the ACT ecosystem. First, the chapter summarizes our embedding of Links and Joints into the ACT ecosystem and details how to get the Link-Joint circuits back into ACT. Lastly, it shows an example of a gate-level Link-Joint network design specified and simulated in ACT.

7.1 *Qnà* — A Shallow Link-Joint Embedding in ACT

The Link-Joint methodology centers around a general and unified abstraction based on the similarities of protocols and asynchronous families. This dissertation showcases the methodology by creating a design flow and test approach around Links and Joints. To achieve this, we utilize elements from ACT — reusing ACT’s application and programming techniques at the top and ACT’s circuit and fabrication techniques at the bottom, as illustrated in Figure 7.1. We use ACT because ACT is the most promising open-source asynchronous tool flow available today. However, **the Link-Joint methodology can be integrated similarly into many asynchronous design flows** to expand the flow’s support for fine-grained mixing and matching of various protocols and circuit families.

Currently, the embedding of the Link-Joint methodology in ACT is shallow. We begin in the ACT ecosystem by using the source languages to specify designs as

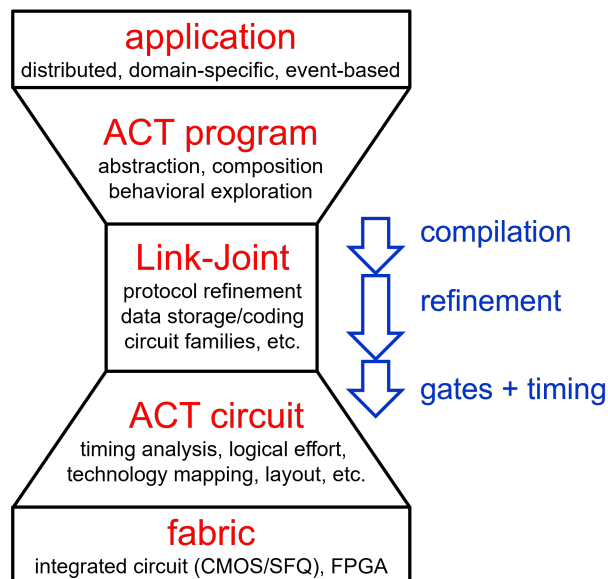


Figure 7.1: Our design and test flow, $\mathcal{Qnà}$, is implemented as a shallow embedding in the ACT ecosystem. All design and test explorations and refinements are done entirely outside of the ACT ecosystem, denoted by the middle part of the hourglass.

programs. We re-engineered the ACT program compiler to generate a hierarchical network of Links and Joints. We use Verilog to model and validate Link-Joint networks and their refinements down to abstract gate-level circuit implementations. Link-Joint network refinement and validation are done outside the ACT ecosystem, in a Link-Joint ecosystem by itself, denoted by the middle of the hourglass in Figure 7.1. The shallow embedding developed in this dissertation enabled us to explore different pieces of our design flow, $\mathcal{Qnà}$, with limited interference from and to the ACT ecosystem. However, to utilize ACT’s fabrication techniques and tools, we need to convert our gate-level specifications into a format that ACT tools can process.

In the future, we aim to natively integrate Links and Joints within the ACT ecosystem by creating a full embedding. This integration would offer both flexibility in asynchronous design and comprehensive design automation.

7.2 Back into ACT with Gates and Timing

ACT uses *Production Rule Set* (PRS) [35,40] and *timing constraints* [1,67] to represent gate-level digital circuits.

7.2.1 Production Rule Set (PRS)

PRS specifies *pull-up and pull-down transistor* networks for CMOS gate implementations. The syntax for PRS is reminiscent of guarded commands [15] and is of the form, $G \rightarrow S$, where G is a Boolean expression and S is a command, typically a signal change that is executed when the Boolean expression is TRUE.

For example, Figure 7.2 presents a two input AND gate using PRS and Verilog specifications. The AND gate is combinational. Its two production rules have complementary Boolean guards for the rising and falling transitions on the AND gate output. ACT automatically generates both production rules when the PRS syntax uses a double arrow (\Rightarrow) instead of a single arrow (\rightarrow) [35].

```

1 // PRS Implementation of a two input AND gate
2 defproc and2 (bool A, B; bool Y){
3     prs {
4         A & B => Y+
5     }
6 }

1 // Verilog Implementation of a two input AND gate
2 module and2 (A, B, Y);
3     input A, B;
4     output Y;
5
6     assign #2 Y = A & B;
7 endmodule

```

Figure 7.2: PRS and Verilog specifications of a two input AND gate.

Figure 7.3 presents a two input *C*-element gate in PRS and Verilog. A *C*-element is state-holding and has non-complementary guards for the rising and falling transitions on the *C*-element output. Thus, specifying both the *C*-element output transitions requires two production rules.

```

1 // PRS Implementation of a two input C-element gate
2 defproc C2 (bool A, B; bool Y){
3     prs {
4         A & B -> Y+
5         ~A & ~B -> Y-
6     }
7 }

1 // Verilog Implementation of a two input C-element gate
2 module C2 (A, B, Y);
3     input A, B;
4     output Y;
5     reg m1;
6     assign Y = m1;
7     always @(A or B) begin
8         if ((A & B) | (!A & !B)) begin
9             #2 m1 = (A && B);
10        end
11    end
12 endmodule

```

Figure 7.3: PRS and Verilog specifications of a two input *C*-element gate.

7.2.2 Timing Constraints

Asynchronous circuits, like synchronous circuits, may be susceptible to delay variations in signal arrival times and timing hazards [66]. One way to understand timing in an asynchronous circuit is through signal transitions, that is, rising (+) or falling (−) transitions within the circuit. These signal transitions are referred to as events. The order of events in a design can be represented as a signal transition graph showing the dependency of events. However, not all event orderings are guaranteed by the design.

In cases where event orderings are not guaranteed, the design must be enhanced with timing constraints.

The conventional use of timing constraints has the notation $e_0 : e_1 < e_2$, where e_0 , e_1 , and e_2 represent signal transitions. In earlier works [22,47,67], this constraint is understood as “after event e_0 , event e_1 must happen before event e_2 .” This dissertation adopts the ACT notation and interpretation for timing constraints, officially referred to as *timing forks* [1,24,37,38]. With the ACT interpretation as a timing fork, the constraint $e_0 : e_1 < e_2$ reads as “after event e_0 , a potential event e_1 , if any, must happen before a potential event e_2 , if any, assuming all potential events happen before the next e_0 event”. Both interpretations see e_0 as a point of divergence, e_1 as the early or fast event, and e_2 as the late or slow event. A timing fork, such as $a+ : b- < c+$, can also be formulated as an error predicate, “if $a+$ is followed by $c+$ and then $b-$, without any intervening instances of $a+$, the timing constraint is violated [24,37]”.

Therefore, to ensure correct operation of the final circuit, these timing constraints need to be statically analyzed during circuit and layout design, and appropriate delays are to be added at specific locations to address the timing issues [67]. Note that signal transition graphs can have cycles. Thus, when analyzing timing in designs in ACT using Cyclone [24], the static timing analyzer available with ACT, it is helpful to indicate the boundary of the signal transition loop (or cycle). The indication is through a loop marker called “*tick*”, which ACT uses during static timing analysis to differentiate subsequent loop iterations [24,37].

There are inherently timing loops in Link-Joint circuits due to the interaction between Links and Joints. Between a Joint having a turn on a Link and relinquishing the turn, there are signal transitions and the signal transitions would occur again as soon as the Joint has the turn on that Link again and then relinquishes the turn once again. Using ACT timing convention, we conservatively assume that most timing

loops pass through the Joint in the Link-Joint circuit, and thus, we “tick”-mark Link-Joint loops at the signal transitions at each Joint’s *MrGO* gate. We provide more details on timing in Link-Joint circuits with an example in the next section.

7.3 Gate-level Link-Joint example in ACT

To illustrate getting a Link-Joint design back into ACT, we use a simple example, *Ring2*, which right-rotates data it receives and passes it along within a ring. Figure 7.4 shows a Link-Joint network for *Ring2*. We have chosen to implement the Links in *Ring2* using the *Click* circuit family and also chosen *2-phase level-signaling bundled-data* protocols for the design. Figure 7.5 shows the design’s gate-level representation and highlights the circuit’s signal names. Note that signals alias each other. For example, *L0.B_me* aliases *J0.in_me* and *L0.A_you* aliases *J1.out_you*.

Figures 7.6 and 7.7 show the PRS specifications of gates hierarchically composed to form modules of Links and Joints used in this *Ring2* example.

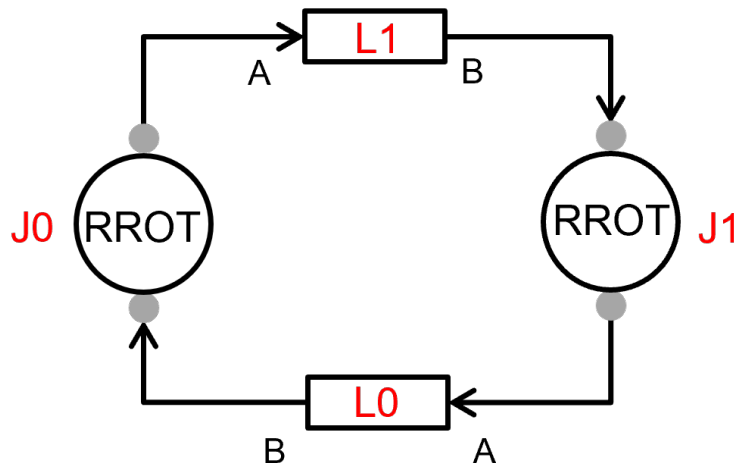


Figure 7.4: Link-Joint network for a 2-stage ring, *Ring2*, with *RRROT* Joints.

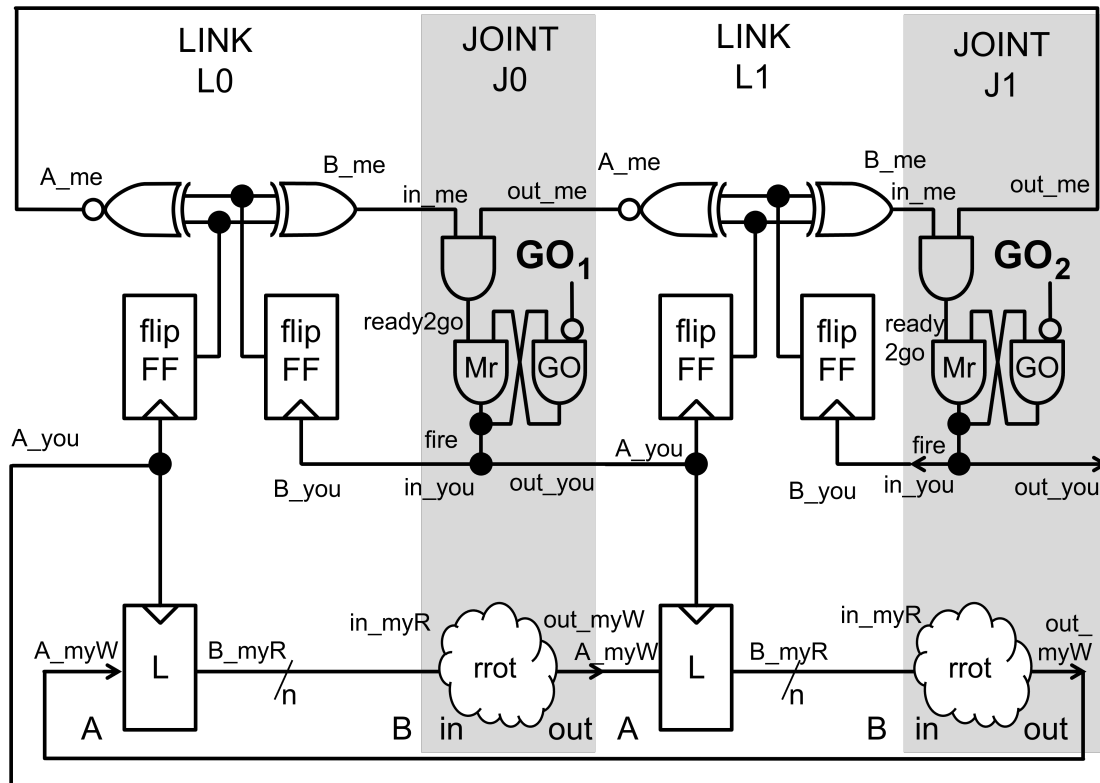


Figure 7.5: Circuit representation of *Ring2* design using *Click* Links and *2-phase level-signaling bundled-data* protocol showing all its signal names.

```

1 defproc inv (bool A; bool Y){ prs { A => Y- } }
2
3 defproc and2 (bool A, B; bool Y){ prs { A & B => Y+ } }
4
5 defproc xor2 (bool A, B; bool Y)
6   { prs { (A & ~B) | (~A & B) => Y+ } }
7
8 defproc xnor2 (bool A, B; bool Y)
9   { prs { (A & B) | (~A & ~B) => Y+ } }
10
11 defproc MrGO (bool start, in, go; bool out){
12   // MrGO // start is there to avoid initial instability
13   prs { in & go & start -> out+
14         ~in      | ~start -> out- }
15 }

```

Figure 7.6: PRS specifications of gates used in Link and Joint modules in *Ring2*.

```

1 defproc slatch (bool d, en, scan_d, scan_en; bool q){
2   // scan latch // use exclusive high en, scan_en
3   spec { exclhi(en, scan_en) }
4   prs { ( d & en) | ( scan_d & scan_en) -> q+
5         (~d & en) | (~scan_d & scan_en) -> q- }
6 }
7
8 defproc sffHI (bool d, en, scan_d, scan_en; bool q){
9   // scan flipflop // use exclusive high en, scan_en
10  bool master;
11  spec { exclhi(en, scan_en) }
12  prs { d & ~en -> master+
13        ~d & ~en -> master-
14        ( master & en) | ( scan_d & scan_en) -> q+
15        (~master & en) | (~scan_d & scan_en) -> q- }
16 }
17
18 defproc sfffHI (bool en, scan_d, scan_en; bool q){
19   // Click's scan flipping FF
20   // use exclusive high en, scan_en
21   bool d;
22   spec { exclhi(en, scan_en) }
23   inv   inv_1(q, d);
24   sffHI sffHI_1(d, en, scan_d, scan_en, q);
25 }

```

Figure 7.7: PRS specifications of gates used in Link and Joint modules in *Ring2*.

7.3.1 Link Implementation for *Ring2*

The Links in the *Ring2* example uses a 2-phase level-signaling bundled-data *Click* implementation. Figure 7.8 gives the ACT-PRS specification with timing constraints typical for a *Click* Link. The timing constraints on lines 19 to 20 in Figure 7.9 ensure mutual exclusivity for *turn* updates through the *xnor2* and *xor2* gates (lines 9 and 10) in the Link. The two timing constraints ensure that upon *A_you+*, both *A_me-* and *B_me+* occur, and that upon *B_you+* both *A_me+* and *B_me-* occur. We visually illustrate the timing fork expressions for the Link as paths shown in Figure 7.9.

```

1 // A 2-phase level-signaling Link in the Click family
2 // to store and transfer 3-bit data from port A to port B
3 defproc L_Click (bool A_you, A_myW[3], B_you, scan_Ah,
4   scan_Bh, scan_AmyW[3], scan_en_data, scan_en_control;
5   bool A_me, B_me, B_myR[3]){
6   bool req, r, a, ack;
7   sfffHI sFFA_req(A_you, scan_Ah, scan_en_control, req);
8   sfffHI sFFB_ack(B_you, scan_Bh, scan_en_control, ack);
9   xnor2  xnorA(req, ack, A_me);
10  xor2   xorB(req, ack, B_me);
11
12  // data storage with scan access
13  slatch slatch_0(A_myW[0], A_you, scan_AmyW[0],
14   scan_en_data, B_myR[0]);
15  slatch slatch_1(A_myW[1], A_you, scan_AmyW[1],
16   scan_en_data, B_myR[1]);
17  slatch slatch_2(A_myW[2], A_you, scan_AmyW[2],
18   scan_en_data, B_myR[2]);
19
20  // Link-related timing constraints:
21  spec { // "turn" updates alternate
22    timing A_you+ : A_me- < B_you*+
23    timing B_you+ : B_me- < A_you*+
24  }
25 }

```

Figure 7.8: ACT-PRS specification for a 2-phase level-signaling bundled-data Click Link with timing constraints typical for Click circuits.

The timing fork $A_you+ : A_me- < B_you*+$, on line 19 of Figure 7.8, is an error predicate. This timing fork expression for Link $L1$ is represented by the red path (both solid and dotted) in Figure 7.9. Recall that in Section 7.2.2 we discussed that Link-Joint timing loop boundaries are “tick”-marked at each Joint’s *MrGO* gate. Thus, a violation of the timing fork could mean either of two things:

1. For Link $L1$, the solid red path, which should be fast from A_you+ to A_me- is actually too slow, meaning a new B_you+ event (from the loop boundary at Joint $J1$) is starting before the current A_me- event is finished.

- For Link $L1$, the dotted red path, which should be slow from A_you+ to B_you+ is actually too fast, meaning a new B_you+ event (from the loop boundary at Joint $J1$) is starting before the current A_me- is done.

In both cases, the *xnor2* gate from $L1$ to $J0$ may receive two input events at the same time, one from each *flipFF*. These now non-mutually exclusive but overlapping input events stall the *xnor2* gate, preventing it from changing its output. As a result, Joint $J0$ believes that it still has the *turn* on port *out*, and so will keep filling *out* with “new” data (duplicate copies) from port *in*. Thus, both outcomes may lead to more data items in the ring up to the point of filling the ring to the brink and deadlock. Both issues can be solved with delay insertion.

The second constraint, on line 20 of Figure 7.8, is similar, except that its two outcomes may lead to fewer data items in the ring up to the point of emptying the

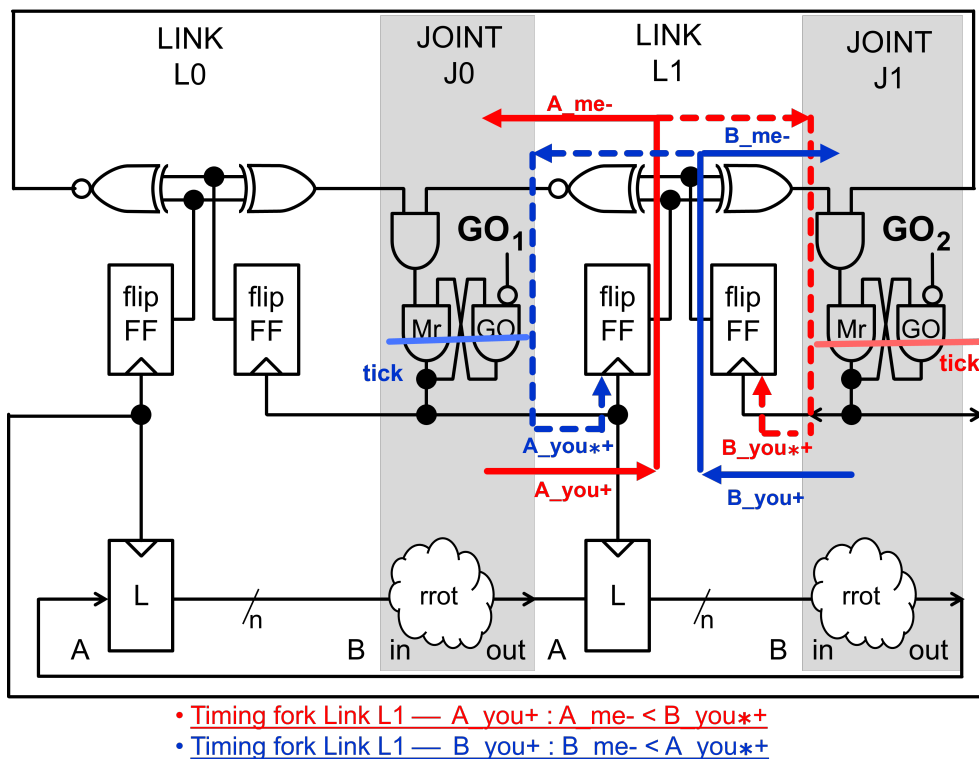


Figure 7.9: Visual depiction of timing forks in the *Click* Link as paths.

ring of data and deadlock. This is possible because Joint $J1$ believes that it still has the *turn* on port *in*, and so will keep providing port *in* with new data without forwarding the previous content, leading to data overwriting.

The $*$ on the late events, $B_you * +$ and $A_you * +$, of the timing fork expressions indicates that the paths from point-of-divergence $A_you +$ to $B_you +$ (red path) and from point-of-divergence $B_you +$ to $A_you +$ (blue path) each go through a loop boundary marker, “*tick*”, in the Joints.

7.3.2 Joint Implementation for *Ring2*

The *RROT* Joints in *Ring2* implement a 2-phase level-signaling protocol. Figure 7.10 defines Joint *RROT* with ACT-PRS specification and timing fork expressions. As discussed in Section 7.2.2, the Link-Joint timing loop boundaries are “*tick*”-marked at each Joint’s *MrGO* gate. The “*tick*” is from *ready2go+* to *fire+*, as specified on line 19 in Figure 7.10.

The timing forks in the Joint ensure that the protocols of the connected Links remain in sync. Because the *RROT* Joints in this *Ring2* example implement a 2-phase protocol, thus, they use an *AND* gate to combine the turns of their connected Links. Consequently, it is possible that, after *fire+*, either *in_me* or *out_me* goes low, resulting in *ready2go-* and consequently *fire-* before both Links have responded to *fire+* event. This possibility would lead to an error, as explained next.

The timing fork $fire+ : in_me- < out_me * +$ on line 21 is an error predicate. We represent the timing fork of Joint $J0$ as the red and blue path in Figure 7.11. The violation of the timing fork could mean either of two things:

1. The solid red path, which should be fast from *fire+* to *in_me-* is actually too slow.

```

1 // A 2-phase level signaling Joint to right rotate 3-bit
  data
2 // from port in to out
3 defproc J_RROT (bool in_me, in_myR[3], out_me, go, start;
4   bool in_you, out_you, out_myW[3]; bool fire){
5   bool ready2go;
6   and2 and2_1(in_me, out_me, ready2go);
7   MrGO mrgo_1(start, ready2go, go, fire);
8   in_you = fire;
9   out_you = fire;
10
11  // data operation
12  out_myW[0] = in_myR[2];
13  out_myW[1] = in_myR[0];
14  out_myW[2] = in_myR[1];
15
16  // Joint-related timing constraints
17  spec {
18    // time ticks at MrGO, from ready2go+ to fire+
19    timing ready2go+ -> fire+
20    // Link protocols remain in sync
21    timing fire+ : in_me- < out_me**
22    timing fire+ : out_me- < in_me**
23  }
24 }

```

Figure 7.10: ACT specification for a 2-phase level-signaling bundled-data *RROT* Joint with timing constraints typical for 2-phase level signaling.

2. The dotted red and blue path, which should be slow from the current *fire+* event to a new *out_me+* event is actually too fast.

These violations mean that port *J0.out* already finished its tasks in the current computation and initiated its tasks in the next computation prematurely. Both error outcomes may lead to deadlock or additional data items in the ring. The second constraint, on line 22 in Figure 7.10, is similar except the error outcomes may lead to deadlock or missing data items in the ring.

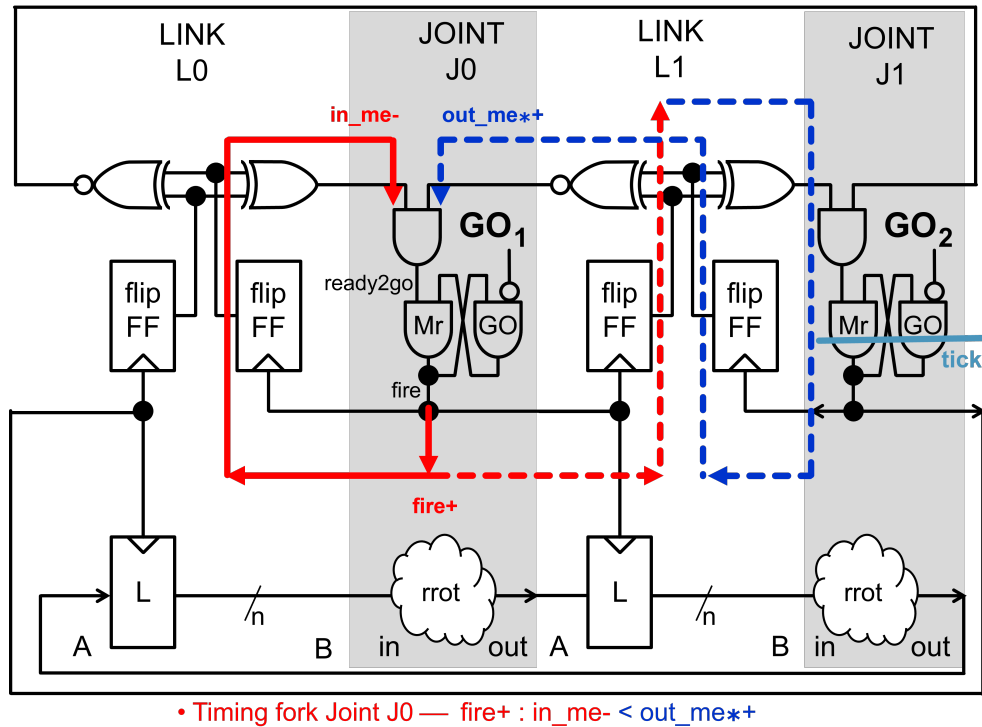


Figure 7.11: Visual depiction of timing constraints in the *RROT* Joint.

7.3.3 Link-Joint network Implementation for *Ring2*

The constraints in Figures 7.8 and 7.10 are local to each process. Due to their similar error outcomes, it will not come as a surprise that all four constraints in Figures 7.8 and 7.10 can be combined into the following constraint pair for each Link-Joint pair: $fire+ : in_me- < B_you * +$ and $fire+ : out_me- < A_you * +$. However, the resulting constraint pair is more restrictive than the four separate constraints and obscures the role that *xnor2* and *xor2* gates play in a Click Link, which is why we opted for four local constraints.

In addition, the entire *Ring2* design also requires timing constraints to govern the interactions among the Links and Joints in the network. Figure 7.12 gives the ACT-PRS specification for *Ring2*, a 2-stage ring structure using two Click Links and two *RROT* Joints with timing constraints governing the handing over of data in the

ring. The timing constraints specify the relation between control versus data paths that start in one Joint and end in the Link following the next Joint. The starting Joint causes the Link between the two Joints to forward data while the next Joint causes the next Link to capture that data.

```

1  //A 2-stage ring structure: L0->J0->L1->J1->L0
2  defproc Ring2 (bool scan_Ah[2], scan_Bh[2], scan_AmyW
   [2][3], scan_en_data, scan_en_control[2], go[2], start)
   {
3  L_Click L[2];
4  J_RROT J[2];
5  (i:2: // external connections
6  L[i].scan_Ah = scan_Ah[i];
7  L[i].scan_Bh = scan_Bh[i];
8  (j:3: L[i].scan_AmyW[j] = scan_AmyW[i][j];)
9  L[i].scan_en_data = scan_en_data;
10 L[i].scan_en_control = scan_en_control[i];
11 J[i].go = go[i];
12 J[i].start = start;
13 // internal connections
14 L[i].A_me = J[i=0?1:i-1].out_me;
15 L[i].A_you = J[i=0?1:i-1].out_you;
16 L[i].B_me = J[i].in_me;
17 L[i].B_you = J[i].in_you;
18 (j:3: L[i].A_myW[j] = J[i=0?1:i-1].out_myW[j];)
19 L[i].B_myR = J[i].in_myR;
20 )
21 // ring related timing constraints:
22 (i:2:
23 (j:3:
24 spec { // lossless data copy and transfer in the
ring
25 timing J[i].fire+ : L[i].A_myW[j] < L[i].A_you*-
26 }
27 )
28 )
29 }

```

Figure 7.12: ACT specification for a 2-phase level-signaling bundled-data 2-stage ring with Joint *RROT* for 3-bit data with timing constraints typical for handing over data in the ring.

The two constraints ensure that no forwarded data are left behind in the computation path — their computed results are captured in the next Link. Because these are timing constraints and not functional constraints, there is no need to specify the data computation. Figure 7.14 at the end of the chapter depicts the timing constraints as colored paths through the circuit.

We simulated *Ring2* with one data element in the ring. To do this, we initialized Link $L[1]$ to be empty and Link $L[0]$ to be full with data value $3'b001$, using the Link scan interface signals, while disabling the two Joints. Then we enabled the Joints to allow the ring to right-rotate and circulate the data. After a while, we disabled Joint

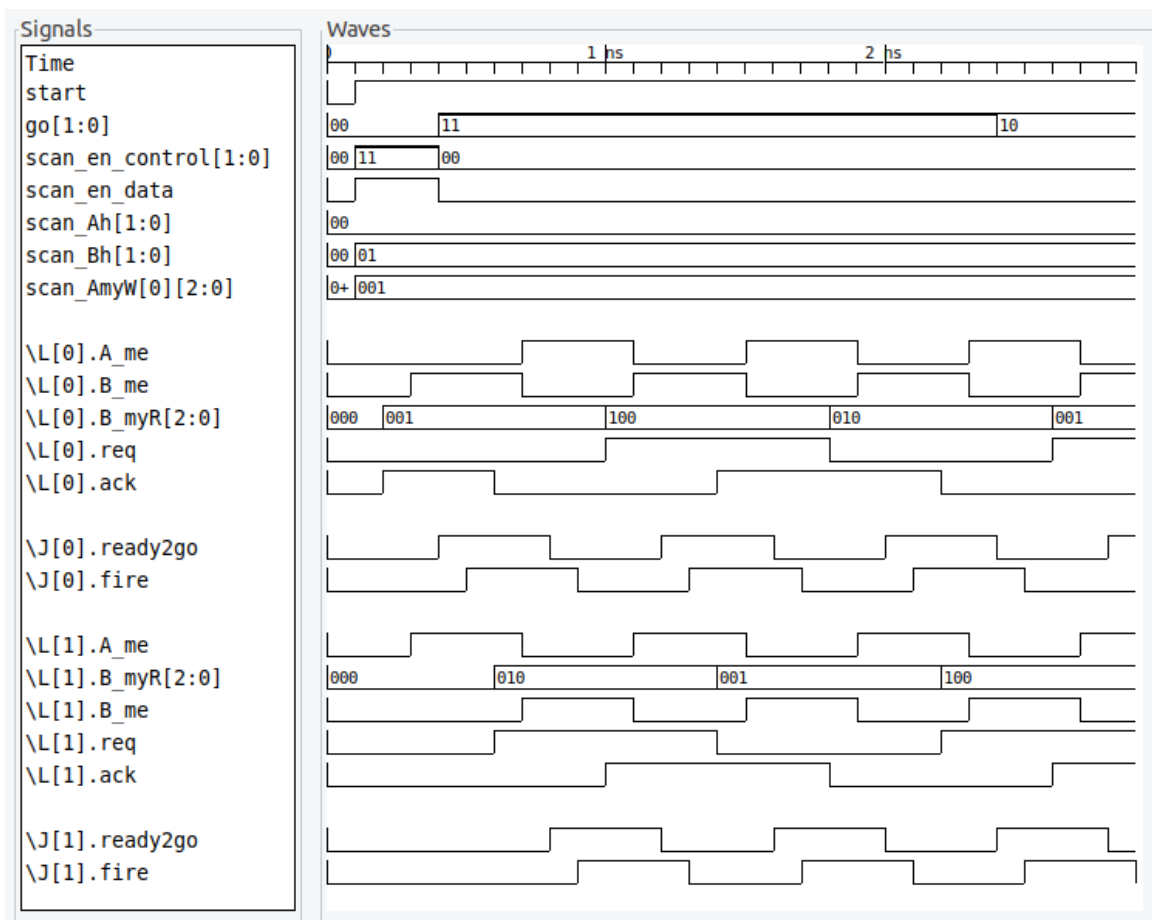


Figure 7.13: Simulation waveform *Ring2*, rotating and circulating one data item, starting with $B_myR[2:0] = 3'b001$ in Link $L[0]$.

$J[0]$ by lowering its go signal, $go[0]$. Figure 7.13 shows the corresponding waveforms. Note the handover of rotated data from $L[0].B_myR[2 : 0]$ to $L[1].B_myR[2 : 0]$ and back — right-rotated from low to high bit and around.

The corresponding ACT test with the initialization and run instructions is in Appendix A, together with an ACT-generated text output file with simulation events to complement the waveforms in Figure 7.13.

7.3.4 Final Notes on Timing Constraints

Timing constraints apply to all asynchronous circuit families, with some families having stricter timing constraints than others. Because we are back in the ACT ecosystem at a low level, we use ACTSIM to simulate our implementations with PRS and timing forks. Our timing constraint definitions for Links and Joints are based on work by Allie Hanson [22] and Hoon Park [46, 47]. Hanson’s work focused on timing constraints for mixed Click and GasP family implementations (to be discussed in the next section). Park determined the timing constraints for asynchronous Click circuits using model checking and tied the verified constraints to the Static Timing Analysis code used then. Verifying and checking the completeness of the timing constraints we present in our design is outside the scope of this dissertation. However, with ACTSIM, we can check the soundness of the specified timing constraints dynamically, at run time.

7.4 Chapter Contributions

This chapter completes the hourglass proposed in Section 1.1.1 — starting and ending in ACT, with refinements and Link-Joint design and test explorations taking place in the middle, outside of ACT. My supervisor and I made a joint effort to translate

our abstract gate implementations in Verilog to PRS specifications. Some timing constraints for Links and Joints were available in the works of Hanson [22], Mettala Gilla [43], and Park [47], but these were from early or even prior to Link-Joint designs. In this chapter, we translated the timing constraints to fit both the new Link-Joint model as defined in this dissertation and the “tick”-based timing fork formalism used by ACT. I implemented and validated the circuit implementations.

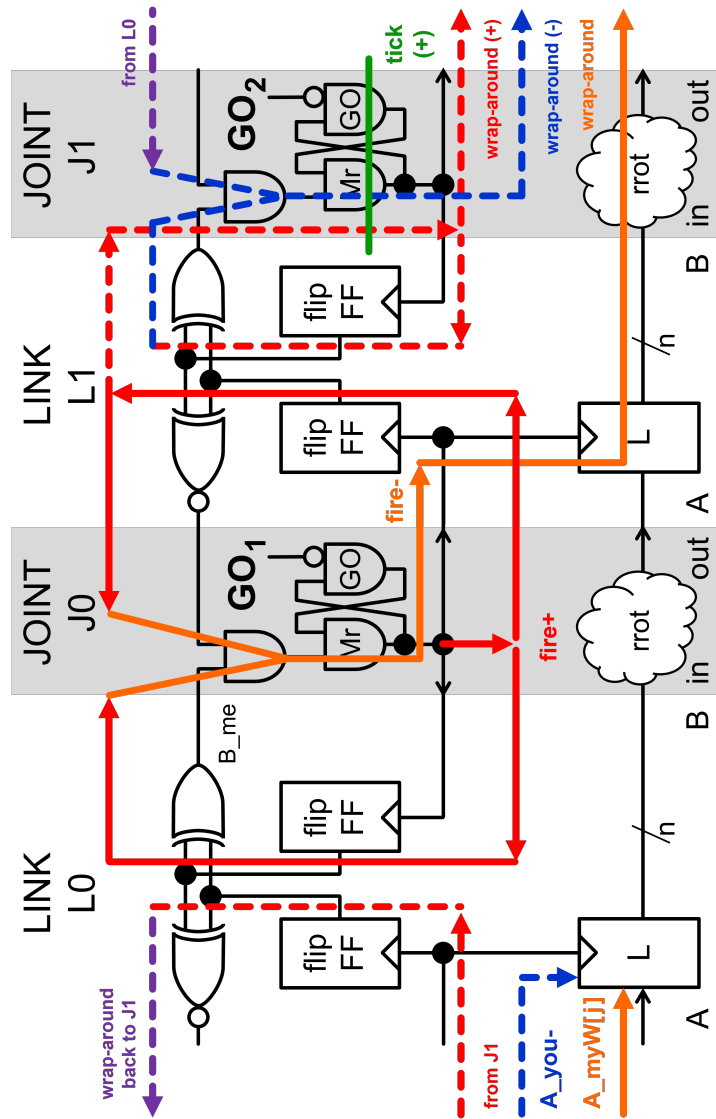


Figure 7.14: Visual depiction of timing paths in the *Ring2* design governing data handover.

Chapter 8: Mixing and Matching Circuit Implementation Styles

Regardless of the protocol, signaling logic, or asynchronous family, Links and Joints provide clear boundaries and well-defined interfaces that ignore differences as much as possible, exploit similarities, and thereby simplify mixing and matching circuit implementation styles. It is helpful to note that the protocols and circuits are alike in every intent but seemingly different in detail. With our approach, designers gain versatility in using different implementation styles at little cost to the overall design effort.

This dissertation has demonstrated the flexibility and generality of Links and Joints by exhibiting the Link-Joint methodology’s support for multiple protocols and circuit families. No one protocol or family is superior to the other; rather, each offers different benefits and can be used depending on the design’s goal. Therefore, we mix and match circuit implementation styles to apply the style appropriate for a design part from all available options.

This chapter showcases striking implementations where we mix these protocols, signaling logics, and circuit families. We also include into our buffet of circuit implementation styles two transition logic families that are yet to be presented in this dissertation but are published in our Festschrift contribution for Steve Furber [63] — Mousetrap [65] and Micropipelines [69].

8.1 Benefits of Various Asynchronous Circuit Implementation Styles

We can briefly summarize some of the finer points of circuit implementation styles used in this dissertation as follows:

Families:

1. *Click*: It is best suited to traditional synchronous evaluation tools for timing analysis and testing because (like synchronous circuits) all its state is stored in flip-flops. It interfaces well with level and transition signaling.
2. *GasP*: It has the highest (control-oriented) throughput due to low logical effort. It works well with level signaling.
3. *Set-Reset*: It is a simpler version of *GasP* but with somewhat higher logical effort and thus somewhat lower (control-oriented) throughput. It works well with level signaling.
4. *Micropipelines and Mousetrap*: They suit *dataflow* programs. They work well with transition signaling and suit data-oriented applications (particularly those with simple control). They have the lowest latency and power because the control operates per transition (as opposed to two transitions for level signaling).
5. *Superconducting*: It has the lowest power (including cooling) and the highest throughput, thanks to superconductivity wires with zero-delay transport.

Signaling:

1. *Level signaling*: It is like Boolean logic and easy to think through.
2. *Transition signaling*: It has fewer events (about half the number of events for control) compared to level signaling but is harder to think through. It requires *xor* gates and latches for complex logic to compare and remember transition type: rising (+) or falling (-).

3. *Pulse signaling*: It is like Boolean logic (at a higher level of abstraction: pulse = TRUE, no-pulse = FALSE). It has fewer events (one pulse versus two level changes) but comes at the cost of more state than level signaling to remember pulse or no-pulse.

Data Encoding:

1. *Bundled data*: This encoding is just as used in synchronous circuits and requires delay matching.
2. *Dual-rail data*: This encoding includes its own readiness signal, so there is less to no need for delay matching. However, it has higher peak power and energy because each bit changes require two actions, set and reset, each time. It is faster for data-dependent operations because the data include their own completion.

Protocol:

1. *2-phase*: It has higher throughput and lower power because there are fewer control events per protocol. It is good for routing.
2. *4-phase*: It has lower throughput than 2-phase, but fewer timing assumptions. It is good for complex local interaction.

Note that a typical combination of *4-phase dual-rail CMOS* will set data in *phase* 1 and 2 and reset data in *phase* 3 and 4. A typical combination of *2-phase dual-rail RSFQ* does not need a data reset *phase* because the pulse is cleared automatically [48, 57]. Section 8.6 details the use of *RSFQ* with *dual-rail* logic in an example. Also, a typical combination of *2-phase bundled data CMOS* does not need to reset data. On average, a *4-phase dual-rail CMOS* implementation uses four times higher

energy compared to *2-phase bundled-data CMOS* implementation because there are two times more control events and two times more data changes.

8.2 Example: A Mixed Ring FIFO In Action

Because protocol and asynchronous family implementation decisions are local refinements in our design flow, exploring mixed implementations is relatively straightforward. Figure 8.1 shows a Link-Joint network for a *6-stage ring FIFO*. This FIFO uses the same interface signals and yet mixes handshaking protocols (*2-phase* and *4-phase*) as well as asynchronous families (*Set-Reset* and *Click*).

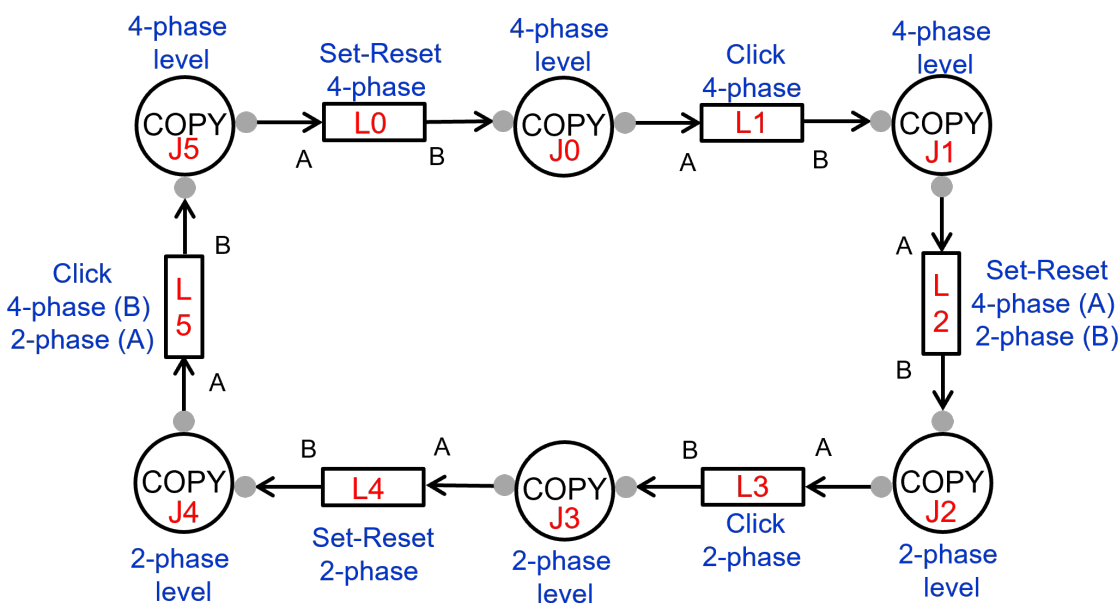


Figure 8.1: Link-Joint network for a 6-stage mixed ring FIFO, using both *2-phase* and *4-phase* protocols as well as *Set-Reset* and *Click* circuit families.

We simulated the functional behavior and throughput of the mixed ring FIFO using the uniform test approach described in Section 6.2. These tests are similar to the tests for a different-sized ring FIFO in RSFQ discussed in Section 6.2 and in the GasP implementation of the Weaver chip [61].

The Canopy graph in Figure 8.2 illustrates the relationship between simulated ring throughput and the number of data items in the ring. The Canopy graph confirms that the mixed ring FIFO operates as expected, although it does not demonstrate that the data are faithfully copied. To assess this, we can refer to the simulated waveforms in the appendix, Figure B.1, which support both the functionality of the mixed design in Figure 8.1 and the relationship between throughput and occupancy in Figure 8.2. Note that detailed *2-phase* and *4-phase* Link and Joint circuit descriptions can be found in the earlier Section 5.3. For circuit details on the mixed *2-phase* and *4-phase* protocols, see the later Section 8.4.

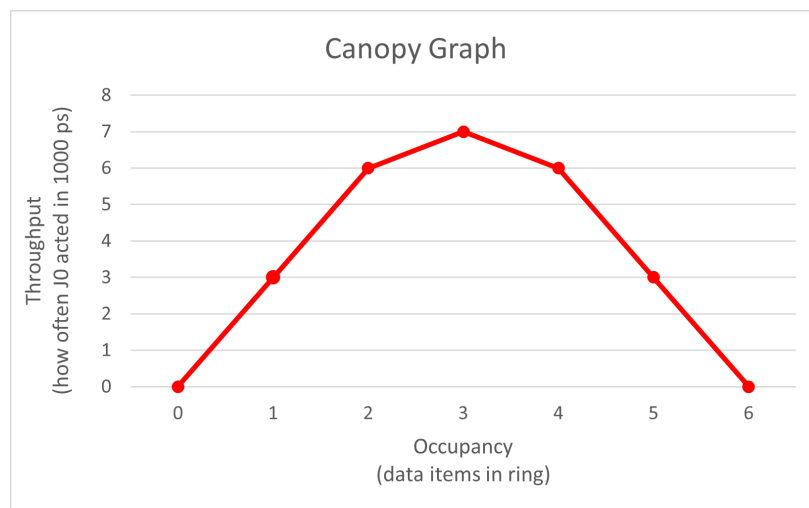


Figure 8.2: Canopy graph of the 6-stage mixed ring FIFO in Figure 8.1.

8.3 Example: Mixing Asynchronous Circuit Families

Because of the state-action separation in the Link-Joint model, differences in asynchronous circuit family implementations are internal to the Links, while the Joints are concerned only about the interface signals and protocols. Therefore, we can swap out Link implementations, which makes mixing and matching different families straightforward.

Figure 8.3 shows a mixed implementation of a simple FIFO with a Set-Reset Link, a *COPY* Joint, and a Click Link, all using *2-phase level-signaling bundled-data* protocols. The signals at the interface for the Links ($myturn(A)$, $yourturn(A)$, $myW(A)$, $myturn(B)$, $yourturn(B)$, and $myR(B)$) remain the same for the different families. Notably, the *Click*-specific signals req and ack are internal to the Click Link on the right and invisible at the Link-Joint interface.

Figure 8.4 shows a *transition-signaling bundled-data* protocol implementation of the same design, but this time using a *Micropipeline* Link (left), a *COPY* Joint (middle), and a *Mousetrap* Link (right). Further details on *transition signaling*, *Micropipeline*, and *Mousetrap* circuits can be found in our Festschrift contribution for Steve Furber [63].

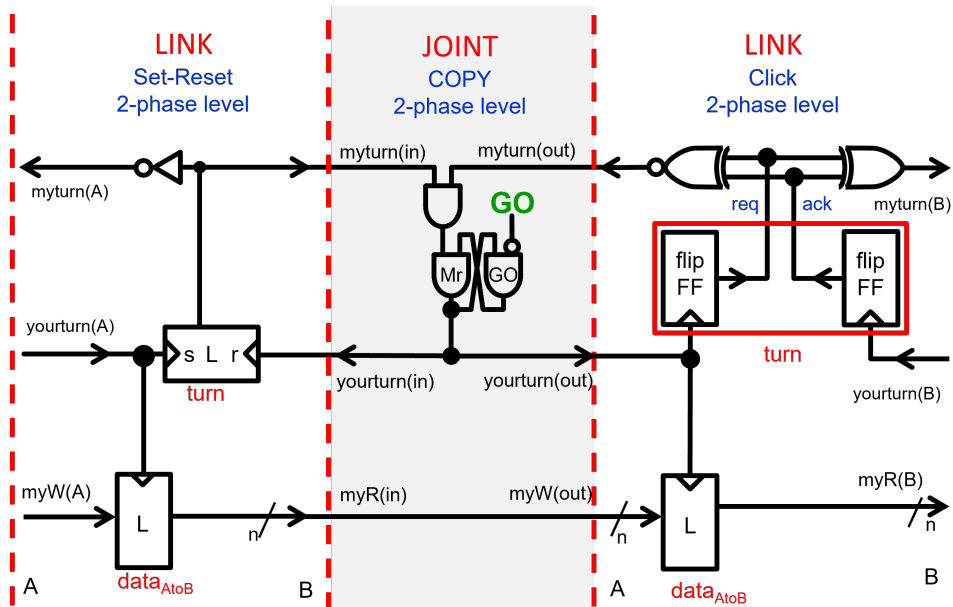


Figure 8.3: Mixed circuit implementation with a *Set-Reset* Link (left), Joint *COPY* (middle), and a *Click* Link (right).

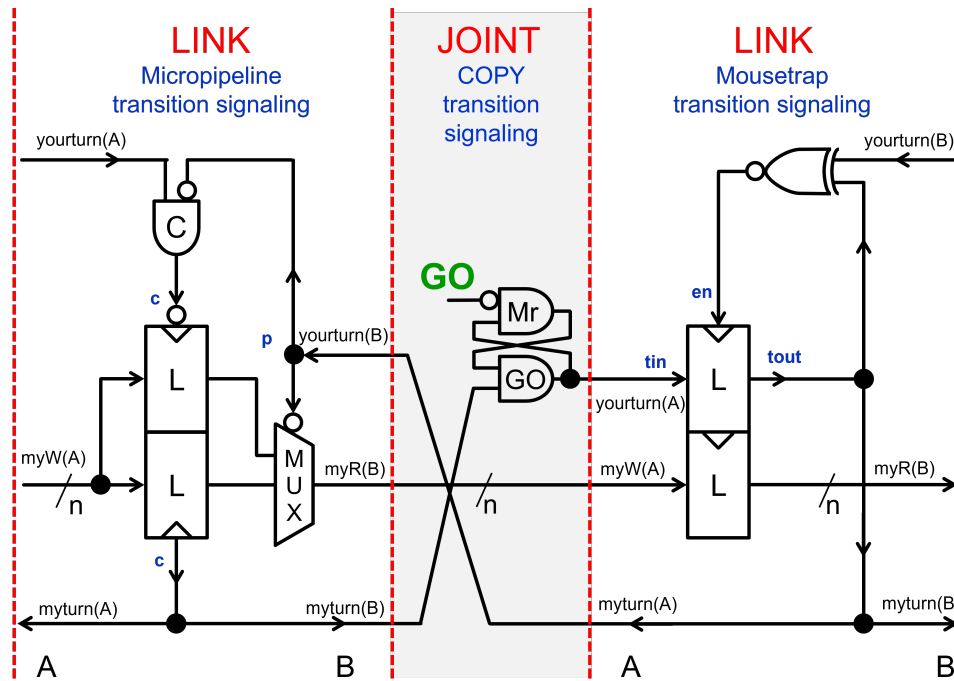


Figure 8.4: Mixed circuit implementation for transition signaling with a *Micropipeline* Link (left), Joint *COPY* (middle), and a *Mousetrap* Link (right).

8.4 Example: Mixing Protocols

Joints have the freedom to choose a protocol independently from their peer Joints. As a result, a Link may use a different protocol at its port *A* than it uses at its port *B*. Figure 8.5 shows three Links in *Click*, *GasP*, and *Set-Reset* using a *2-phase bundled-data level-signaling* protocol at port *A* and a *4-phase* counterpart at port *B*. These Links combine the 2-phase and 4-phase protocol versions for the circuit families presented earlier in Section 5.3 and have been published at the ASYNC 2023 conference [17].

Figure 8.6 mixes a corresponding *4-to-2 phase GasP* Link and a *2-to-4 phase Click* Link to connect a *2-phase* Joint with a *4-phase* Joint. The two Joints were presented earlier in Section 5.3, Figure 5.13.

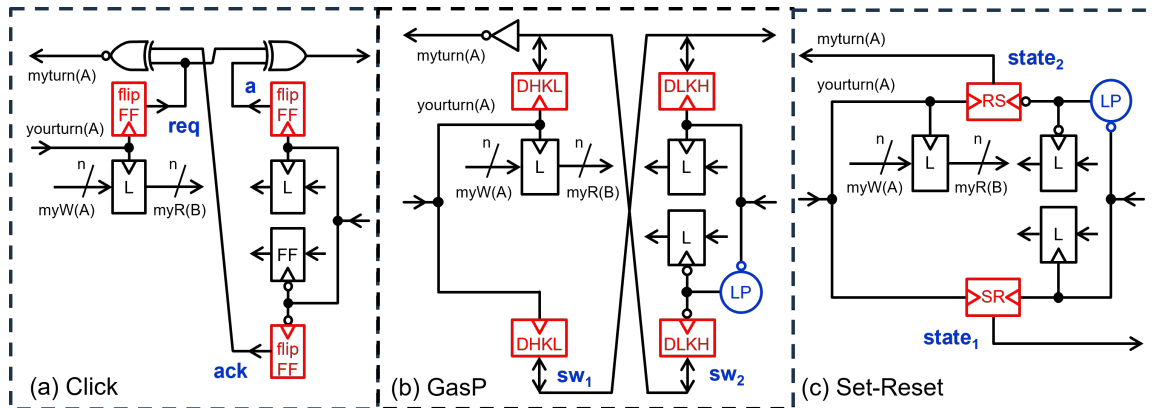


Figure 8.5: Bidirectional Links in circuit families (a) *Click*, (b) *GasP*, and (c) Set-Reset using *2-phase* at Link port A, and *4-phase* at Link port B.

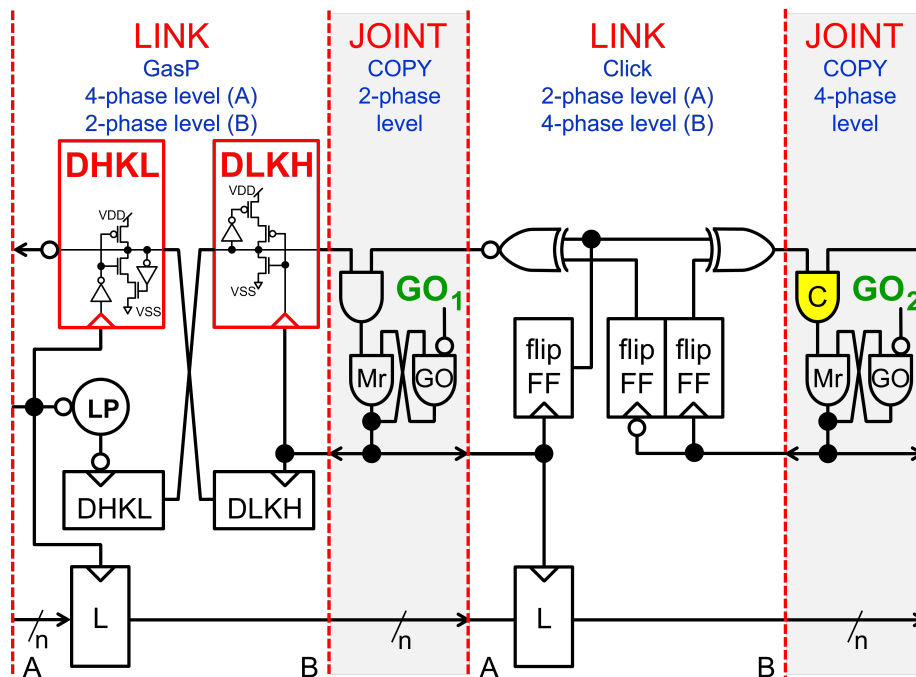


Figure 8.6: Mixed circuit implementation where a *2-phase* Joint (left) and a *4-phase* Joint (right) are connected with a *4-to-2 phase GasP* Link and a *2-to-4 phase Click* Link. This Link-Joint FIFO also functions correctly when connected as a ring FIFO.

8.5 Example: Mixing Signaling Logic

We can also mix different signaling logics. *Click*'s flipping flipflop, *flipFF*, introduced by Ad Peeters et al. [50] is ideal for connecting *transition* and *level* signaling. In

Figures 8.7 and 8.8, we use the *flipFF* outputs as both a *transition* signal and a *level* signal. For the transition-based part of the design, the *flipFF* output is a transition, but the level-based part sees it as a level signal.

Figure 8.7 mixes a *transition to 2-phase level signaling* Link, *L1*, and a *2-phase level to transition signaling* Link, *L2*, to connect a *2-phase level-signaling* Joint, *J1*, with a *2-phase transition-signaling* Joint, *J2*. Link *L1* combines *Mousetrap* with *Click*. Link *L2* uses only *Click* elements. Figure 8.8 is similar, but instead of *2-phase level signaling*, it uses *4-phase level signaling* in Joint *J1* and the connecting Links.

Each of the two Link-Joint FIFO designs also functions correctly when connected as a ring FIFO. Both designs were published in our Festschrift contribution for Steve Furber [63].

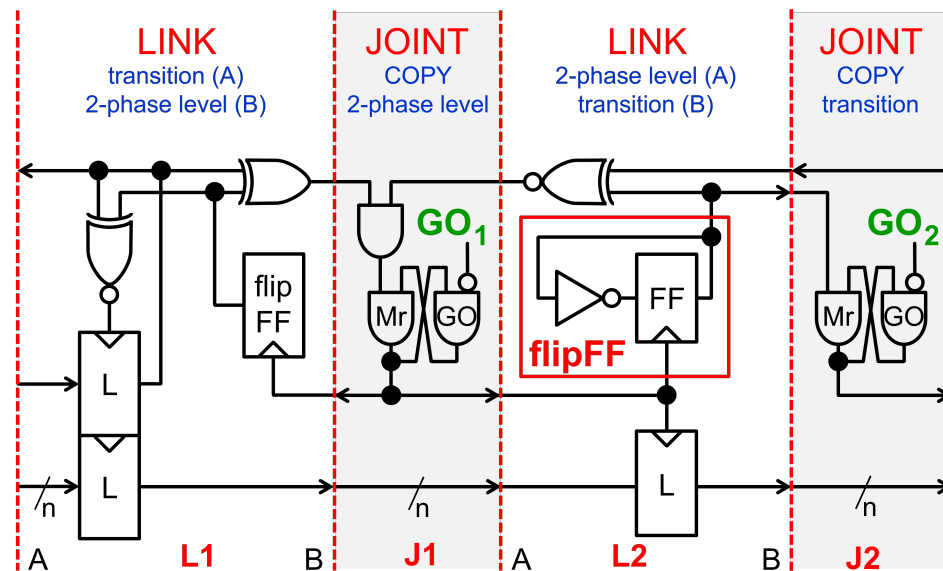


Figure 8.7: Mixed circuit implementation where a *2-phase level-signaling* Joint (left) and a *transition* Joint (right) are connected with a *transition to 2-phase level signaling* Link, *L1*, based on *Mousetrap* and *Click*, and a *2-phase level to transition signaling* Link, *L2*, based on *Click*.

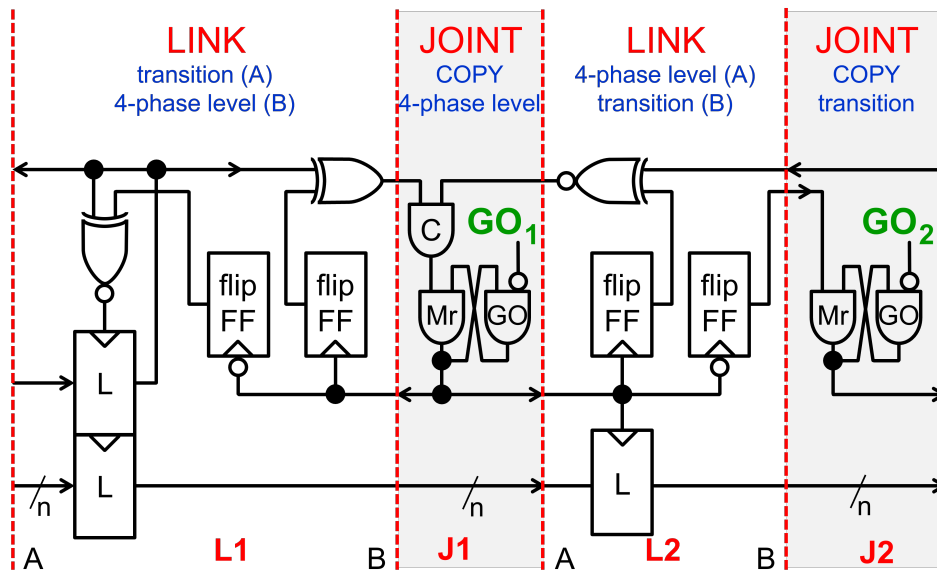


Figure 8.8: Mixed circuit implementation where a *4-phase level-signaling* Joint (left) and a *transition* Joint (right) are connected with a *transition to 4-phase level signaling* Link, *L1*, based on *Mousetrap* and *Click*, and a *4-phase level to transition signaling* Link, *L2*, based on *Click*.

8.6 Matching Implementations to Links and Joints

Mixing different Link-Joint implementation styles is relatively easy, as the previous sections demonstrate. But what can we do if the asynchronous implementation is not formulated in terms of Links and Joints?

This section shows how we match an existing asynchronous design to a version using the Link-Joint model. The design, shown in Figure 8.9, uses *dual-rail data encoding* and *superconducting RSFQ pulse logic*. It is based on the clocked and pipelined dual-rail adder example that we published in the Transactions on Applied Superconductivity [57], but it replaces the clock with an asynchronous protocol. To replace the clock, the asynchronous protocol includes logic to detect when computed data (1) are complete, (2) may be propagated to the following pipeline stages, and (3) may be overwritten.

Intuitively, the gates behave as follows.

- The *C-element* or *rendezvous* in Figure 8.10(a) produces a pulse on its output when both input signals have a pulse. The input pulses can arrive in any order. While producing the output pulse, the *C-element* clears its input pulses.
- The *OR gate* or *merge* in Figure 8.10(b) receives mutually exclusive input pulses. It produces one output pulse per input pulse (while clearing the input pulse).
- The *SPLIT* or *fork* in Figure 8.10(c) provides “fanout” by taking an input pulse and producing a pulse on each output (while clearing the input pulse).
- The 1×2 *rendezvous gate* in Figure 8.10(d) combines a pulse on its dual-rail data input pair, i.e., a pulse on either the *T* or the *F* input, with a pulse on its enable input, *en*. The 1×2 gate stores its data input until the enable pulse has arrived. When both the data and an enable pulse have arrived, it copies its dual-rail input to its dual-rail output pair. While producing the output pair, it clears the input data and enable pulses.
- The *combinational logic (CL)* in Figure 8.10(e) represents a dual-rail data computation, such as the bit-wise full adder in [48]. It uses dual-rail input pairs, with two wires per bit, and produces dual-rail output pairs. We assume that a CL computation leaves no pulses behind [57].

With this intuitive gate behavior, we can now explain the asynchronous protocol in Figure 8.9. The CL computation takes dual-rail inputs and generates dual-rail outputs. The 1×2 gates store any CL computed data, e.g., $myW(out1)[T,F]$, as soon as these become available. But the 1×2 gates propagate stored data **only** when there is space to receive that data. The C-element that drives the enable input of each 1×2 gate detects if there is space for both data items. There is space when all

1×2 gates in the next pipeline stages, which are of the same form, have propagated their results. The OR gate in Figure 8.9 checks propagation for one 1×2 gate (in the current pipeline stage). Another OR gate can be added for the other 1×2 gate, but this may be unnecessary if the checked 1×2 gate stores the last generated CL result.

Now that we understand the design at the protocol level, we are able to partition it into Links and Joints. Joints compute, so CL goes into a Joint. Links store, so the 1×2 gates go into Links. The resulting partition follows in Figure 8.11. For initialization and test, this Link-Joint partition can be enhanced with a MrGO gate after the C-element in the Joint and with scan access to the 1×2 gates in the Links.

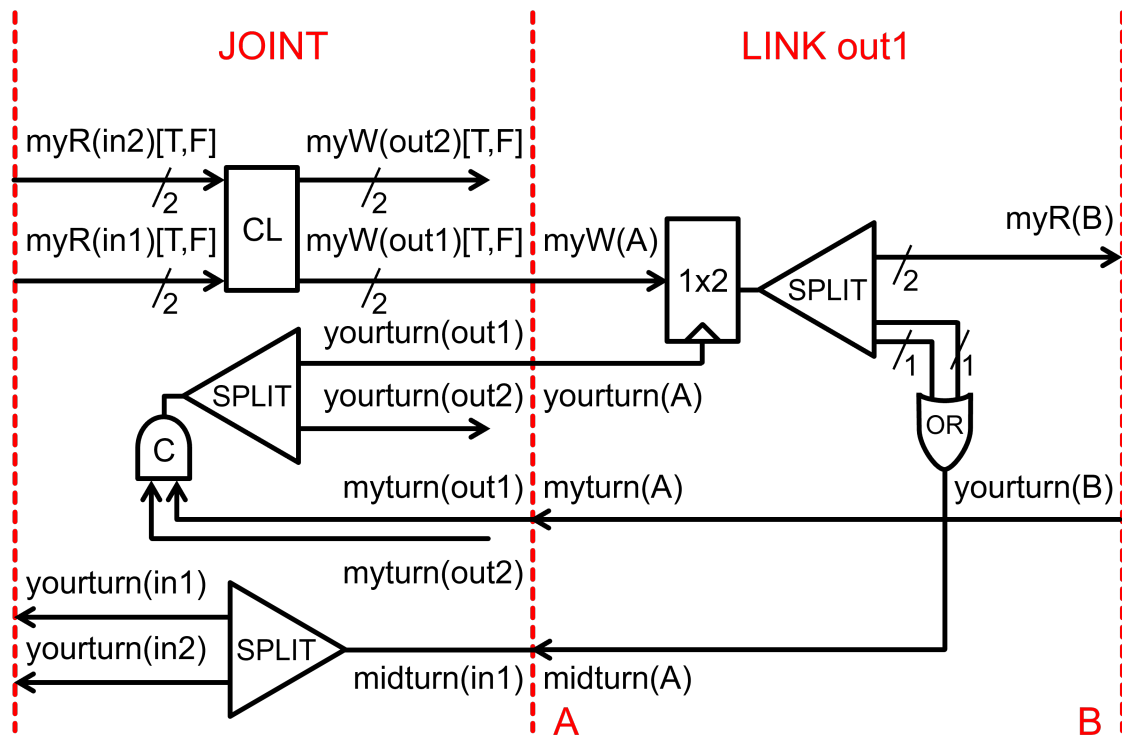


Figure 8.11: Link-Joint partition for the design in Figure 8.9.

8.7 Chapter Contributions

We showcased the ease and generality of mixing and matching different protocols, signaling logics, and circuit families in the Link-Joint model. Mixing implementations not only encourages design reuse and streamlines collaboration between designers with different preferences but also enables designers to choose the best solution for each design part. The examples shown in this chapter extend the work done on mixing circuit families for one type of protocol, based on *2-phase level-signaling bundled-data* [43, 60, 62]. We go beyond this by also mixing protocols and signaling logic. The key takeaway is that mixing implementation styles bridges the implementation gaps for collaboration and design reuse. Though this chapter is a joint effort of my supervisor and me, I implemented and validated the circuit implementations.

Chapter 9: Conclusion and Future Work

The work presented in this dissertation centers around the objective of *making it easy to insert asynchrony appropriate for each design part*. In particular, it focused on the design and test of asynchronous circuits through a general and unified abstraction, *Links and Joints*. We built and presented a design and test flow, *Qnà*, around Links and Joints, combining design automation with Link-Joint flexibility. Our design flow is made by embedding the Link-Joint methodology in an existing and well-used design flow, *ACT (Asynchronous Circuit Toolkit)*. We compile ACT programs with data- and control-flow to circuit-neutral Link-Joint networks, which are refined into circuits.

The approach presented in this dissertation offers the benefits of **flexibility** and **generality**. By using circuit-neutral Links and Joints, we bind circuit implementation decisions as late as possible. This approach allows us to go from a single high-level program to multiple circuit implementations through the same Link-Joint network. It also serves us in testing and debugging, such that we can create any starting state and condition for our designs, pause or start system actions, and observe or alter the state in the design, all from program level to Links and Joints to circuit level, using a uniform test approach. Implementation decisions can also be isolated; for example, each Link has the freedom to choose its own circuit family, and each Joint has the freedom to choose its own communication protocol.

Here is a list of communication protocols, signaling logics, data encodings, and circuit families this work supports or that we are currently developing:

- 2- and 4-phase protocols, level- and pulse- and transition-signaling logic, bundled and dual-rail data,

- Click, GasP, Set-Reset, Mousetrap, Micropipelines, RSFQ.

The work in this dissertation accomplished the following.

1. It illustrates how the Link-Joint methodology can be integrated into a design flow.
2. It showcases the generality and flexibility of the Link-Joint methodology in supporting widely used communication protocols and asynchronous circuit families.
3. It showcases a uniform test and debug strategy that can be translated from one abstraction level to another.
4. It facilitates and showcases the ease of mixing and matching different implementation styles so that designers can select the best protocol, signaling logic, and circuit family for each design part.

Our codebase for *Qnà*, that is, our compiler codebase, abstract Verilog implementations for a Link and several Joint types, and gate implementation of Links and Joints using several protocols and different circuit families, is accessible at: <https://arc.cecs.pdx.edu/code/> [18].

9.1 Future Research Directions

The work in this dissertation has provided a structured route to scalable, accessible, and easy use of Links and Joints. However, realistic widespread use of the methodology still requires more automation and validation. Currently, the embedding of Links and Joints in the ACT ecosystem is shallow. We believe that having Links and Joints natively integrated into ACT would benefit both ACT – by increasing the flexibility and ease of inserting asynchrony appropriate for each design part – and the Link-Joint

methodology – by increasing its viability and user base through the ACT ecosystem and making more tools that are available in ACT also available to Links and Joints.

As mentioned in Chapter 4, more language optimization would be beneficial for more efficient circuits through compilation. Currently, our compilation results are based on the program’s quality, with optimizations done as refinements on the Link-Joint network. The Yale ACT research group is presently working on CHP language optimizations. We believe that as long as the same set of language constructs are used to specify the program, our compiler can seamlessly work with the optimized program. However, if new language constructs are introduced, our compiler needs to be updated.

The test and debug approach presented in this dissertation can be extended further to support more test generation. In addition, to help designers make more informed decisions, design, as well as test and debug, would benefit from tool support for profiling and for graphical visualization of both design and simulations, like those developed for Balsa [3, 26].

The multiple specification models at different levels of abstraction presented in this dissertation would benefit from equivalence checking and from validating translations of higher level specifications to lower level implementations. There are initial results by Cuong Chau [8] comparing the Link and Joint model to the circuit-level communication model. More work must be done to tie all the abstraction levels together formally.

Bibliography

- [1] Samira Ataei, Wenmian Hua, Yihang Yang, Rajit Manohar, Yi-Shan Lu, Jiayuan He, Sepideh Maleki, and Keshav Pingali. An Open-Source EDA flow for Asynchronous Logic. *IEEE Design & Test*, 38(2):27–37, 2021.
- [2] Andrew Bardsley. *Implementing Balsa Handshake Circuits*. PhD thesis, The University of Manchester, United Kingdom, 2000.
- [3] Andrew Bardsley and DA Edwards. The Balsa Asynchronous Circuit Synthesis System. In *Forum on Design Languages*, volume 224, 2000.
- [4] Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, United States, 1991.
- [5] Steven M. Burns. Automated Compilation of Concurrent Programs into Self-Timed Circuits. Master’s thesis, California Institute of Technology, 1987.
- [6] Steven M. Burns and Alain J. Martin. Syntax-directed Translation of Concurrent Programs into Self-Timed Circuits. In *Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.
- [7] Tony Bybell. GTKWave. <http://gtkwave.sourceforge.net/>, 2023. Version 3.3.114.
- [8] Cuong Chau, Warren A Hunt, Matt Kaufmann, Marly Roncken, and Ivan Sutherland. A Hierarchical Approach to Self-Timed Circuit Verification. In *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 105–113. IEEE, 2019.
- [9] Tam-Anh Chu. *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.
- [10] IEEE Standard Committee et al. IEEE Std. 1149.1-2001 IEEE Standard Test Access Port and Boundary Scan Architecture.
- [11] J. Cortadella, A. Kondratyev, L. Lavagno, K. Lwin, and C. Sotiriou. From Synchronous to Asynchronous: An Automatic Approach. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, volume 2, pages 1368–1369 Vol.2, 2004.

- [12] J. Cortadella, A. Kondratyev, L. Lavagno, and C.P. Sotiriou. Desynchronization: Synthesis of Asynchronous Circuits From Synchronous Specifications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):1904–1921, 2006.
- [13] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers. *IEICE Transactions on Information and Systems*, 80(3):315–325, 1997.
- [14] Al Davis and Steven M Nowick. An Introduction to Asynchronous Circuit Design. *The Encyclopedia of Computer Science and Technology*, 38:1–58, 1997.
- [15] Edsger W Dijkstra. Guarded commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, 1975.
- [16] Ebelechukwu Esimai and Marly Roncken. Flexible Active–Passive and Push–Pull Protocols. *IEEE Embedded Systems Letters*, 14(3):139–142, 2022.
- [17] Ebelechukwu Esimai and Marly Roncken. Flexible Compilation and Refinement of Asynchronous Circuits. In *2023 28th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 109–119. IEEE, 2023.
- [18] Ebelechukwu Esimai and Marly Roncken. *Qnà* Codebase. <https://arc.cecs.pdx.edu/code/>, 2024.
- [19] Geeksforgeeks. Unit Testing – Software Testing. <https://www.geeksforgeeks.org/unit-testing-software-testing>, 2023. Accessed: 2024-03-19.
- [20] Gennette D Gill. *Analysis and Optimization for Pipelined Asynchronous Systems*. PhD thesis, The University of North Carolina at Chapel Hill, 2010.
- [21] Dylan Hand, Matheus Trevisan Moreira, Hsin-Ho Huang, Danlei Chen, Frederico Butzke, Zhichao Li, Matheus Gibiluka, Melvin Breuer, Ney Laert Vilar Calazans, and Peter A Beerel. Blade – A Timing Violation Resilient Asynchronous Template. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 21–28. IEEE, 2015.
- [22] Alexandra Hanson. Facilitating Mixed Self-Timed Circuits. Bachelor’s Thesis, Portland State University, 2020.
- [23] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

- [24] Wenmian Hua, Yi-Shan Lu, Keshav Pingali, and Rajit Manohar. Cyclone: A Static Timing and Power Engine for Asynchronous Circuits. In *2020 26th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 11–19. IEEE, 2020.
- [25] Henrik Hulgaard, Steven M Burns, and Gaetano Borriello. Testing Asynchronous Circuits: A survey. *Integration*, 19(3):111–131, 1995.
- [26] Lilian Janin. *Simulation and Visualisation for Debugging Large Scale Asynchronous Handshake Circuits*. PhD thesis, The University of Manchester, United Kingdom, 2005.
- [27] Gleb Krylov. *Design Methodologies for Single Flux Quantum VLSI Circuits*. PhD thesis, University of Rochester, 2021.
- [28] Konstantin K Likharev and Vasilii K Semenov. RSFQ Logic/Memory Family: A new Josephson-Junction Technology for Sub-Terahertz-Clock-Frequency Digital Systems. *IEEE Transactions on Applied Superconductivity*, 1(1):3–28, 1991.
- [29] INMOS Limited. *Occam 2 reference manual*. Prentice Hall, 1988.
- [30] Rasmus Madsen. Desynchronization of digital circuits. Master’s thesis, Technical University of Denmark, 2011.
- [31] M Maezawa, I Kurosawa, M Aoyagi, H Nakagawa, Y Kameda, and T Nanya. Rapid Single-Flux-Quantum Dual-Rail Logic for Asynchronous Circuits. *IEEE Transactions on Applied Superconductivity*, 7(2):2705–2708, 1997.
- [32] Rajit Manohar. ACT Hardware Description Language Documentation. <https://avlsi.csl.yale.edu/act>, 2018.
- [33] Rajit Manohar. ACT Tools. <https://github.com/asyncvlsi/>, 2018.
- [34] Rajit Manohar. The CHP sublanguage. <https://avlsi.csl.yale.edu/act/doku.php?id=language:langs:chp>, 2023.
- [35] Rajit Manohar. The PRS sublanguage. <https://avlsi.csl.yale.edu/act/doku.php?id=language:langs:prs>, 2023.
- [36] Rajit Manohar. The Dataflow sublanguage. <https://avlsi.csl.yale.edu/act/doku.php?id=language:langs:dflow>, 2023.
- [37] Rajit Manohar. The SPEC sublanguage. <https://avlsi.csl.yale.edu/act/doku.php?id=language:langs:spec>, 2024. Timing Constraints and Directives.
- [38] Rajit Manohar and Yoram Moses. Timed Signalling Processes. In *2023 28th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 10–19, 2023.

- [39] Alain J Martin. Compiling Communicating Processes into Delay-Insensitive VLSI Circuits. *Distributed computing*, 1:226–234, 1986.
- [40] Alain J Martin. Formal Program Transformations for VLSI Circuit Synthesis. In Edsger W. Dijkstra, editor, *Formal Development Programs and Proofs*, pages 59–80. Addison Wesley, 1989.
- [41] Alain J Martin. Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*. Addison Wesley, 1990.
- [42] Alain J Martin. *Synthesis of Asynchronous VLSI Circuits*. California Institute of Technology, Computer Science Department, 1993.
- [43] Swetha Mettala Gilla. *Silicon Compilation and Test for Dataflow Implementations in GasP and Click*. PhD thesis, Portland State University, 2018.
- [44] Mika Nystroem. *Asynchronous Pulse Logic*. PhD thesis, California Institute of Technology, Pasadena, California, US, 2001.
- [45] Heechun Park and Taewhan Kim. Hybrid Asynchronous Circuit Generation Amenable to Conventional EDA Flow. *Integration*, 64:29–39, 2019.
- [46] Hoon Park. *Formal Modeling and Verification of Delay-Insensitive Circuits*. PhD thesis, Portland State University, 2015.
- [47] Hoon Park, Anping He, Marly Roncken, Xiaoyu Song, and Ivan Sutherland. Modular Timing Constraints for Delay-Insensitive Systems. *Journal of Computer Science and Technology*, 31:77–106, 2016.
- [48] Priyadarsan Patra, Stanislav Polonsky, and Donald S Fussell. Delay Insensitive Logic for RSFQ Superconductor Technology. In *Proceedings Third International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 42–53. IEEE, 1997.
- [49] Ad Peeters. *Single-Rail Handshake Circuits*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 1996.
- [50] Ad Peeters, Frank Te Beest, Mark De Wit, and Willem Mallon. Click Elements: An Implementation Style for Data-Driven Compilation. In *2010 IEEE Symposium on Asynchronous Circuits and Systems*, pages 3–14. IEEE, 2010.
- [51] Luis A. Plana. *Contributions to the Design of Asynchronous Macromodular Systems*. PhD thesis, Columbia University, New York, United States, 1998.

- [52] Ivan Poliakov, Danil Sokolov, and Andrey Mokhov. Workcraft: A Static Data Flow Structure Editing, Visualisation and Analysis Tool. In Jetty Kleijn and Alex Yakovlev, editors, *Petri Nets and Other Models of Concurrency – ICATPN 2007*, pages 505–514, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [53] Marly Roncken. Partial Scan Test for Asynchronous Circuits illustrated on a DCC Error Corrector. In *Proceedings of 1994 IEEE Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 247–256, 1994.
- [54] Marly Roncken. Defect-oriented Testability for asynchronous ICs. *Proceedings of the IEEE*, 87(2):363–375, 1999.
- [55] Marly Roncken, Emile Aarts, and Wim Verhaegh. Optimal Scan for Pipelined Testing: An Asynchronous Foundation. In *Proceedings International Test Conference 1996. Test and Design Validity*, pages 215–224, 1996.
- [56] Marly Roncken and Eric Bruls. Test Quality of Asynchronous Circuits: A Defect-Oriented Evaluation. In *Proceedings International Test Conference 1996. Test and Design Validity*, pages 205–214. IEEE, 1996.
- [57] Marly Roncken, Ebelechukwu Esimai, Vivek Ramanathan, Warren A Hunt, and Ivan Sutherland. State Access for RSFQ Test and Analysis. *IEEE Transactions on Applied Superconductivity*, 33(5):1–7, 2023.
- [58] Marly Roncken, Ebelechukwu Esimai, and Ivan Sutherland. Async 2022 summer school: Links and Joints: behavioral design. https://avlsi.csl.yale.edu/act/lib/exe/fetch.php?media=summer2022:05_linkjoint1.pdf, 2022.
- [59] Marly Roncken, Ebelechukwu Esimai, and Ivan Sutherland. Async 2022 summer school: Links and Joints: gate-level design. https://avlsi.csl.yale.edu/act/lib/exe/fetch.php?media=summer2022:11_linkjoint_part2_async_summerschool_2022_handout_4pp.pdf, 2022.
- [60] Marly Roncken, Swetha Mettala Gilla, Hoon Park, Navaneeth Jamadagni, Chris Cowan, and Ivan Sutherland. Naturalized Communication and Testing. In *2015 21st IEEE International Symposium on Asynchronous Circuits and Systems*, pages 77–84. IEEE, 2015.
- [61] Marly Roncken and Ivan Sutherland. Design and Test of High-Speed Asynchronous Circuits. In J Di and SC Smith, editors, *Asynchronous Circuit Applications*, pages 113–171. Inst. Eng. Technol.(IET), 2020.
- [62] Marly Roncken, Ivan Sutherland, Chris Chen, Yong Hei, Warren Hunt, Cuong Chau, Swetha Mettala Gilla, Hoon Park, Xiaoyu Song, Anping He, and Hong Chen. How to think about self-timed systems. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 1597–1604. IEEE, 2017.

- [63] Marly Roncken, Ivan Sutherland, and Ebelechukwu Esimai. Micropipelines united. In *We're going to Need a Bigger Computer - Essays dedicated to Steve Furber on the occasion of his retirement. At Last*. University of Manchester Press Unit, 2024.
- [64] Feng Shi. *Simulating and Testing Asynchronous Circuits*. PhD thesis, Yale University, United States, 2007.
- [65] Montek Singh and Steven M Nowick. MOUSETRAP: High-Speed Transition-Signaling Asynchronous Pipelines. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(6):684–698, 2007.
- [66] Jens Sparsø. *Introduction to Asynchronous Circuit Design*. DTU Compute, Technical University of Denmark, 2020.
- [67] Ken S Stevens, Ran Ginosar, and Shai Rotem. Relative Timing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(1):129–140, 2003.
- [68] Ivan Sutherland and Scott Fairbanks. GasP: A minimal FIFO control. In *Proceedings Seventh International Symposium on Asynchronous Circuits and Systems. ASYNC 2001*, pages 46–53. IEEE, 2001.
- [69] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, 1989.
- [70] John Teifel and Rajit Manohar. Static Tokens: Using Dataflow to Automate Concurrent Pipeline Synthesis. In *Proceedings of the 10th International Symposium on Asynchronous Circuits and Systems, 2004.*, pages 17–27. IEEE, 2004.
- [71] CH Van Berkel, Mark B Josephs, and Steven M Nowick. Applications of Asynchronous Circuits. *Proceedings of the IEEE*, 87(2):223–233, 1999.
- [72] Kees van Berkel. *Handshake Circuits: an Intermediary between Communicating Processes and VLSI*. PhD thesis, Technical University Eindhoven, The Netherlands, 1992.
- [73] Kees Van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, Frits Schalijs, and Ad Peeters. Asynchronous Circuits for Low Power: A DCC Error Corrector. *IEEE Design & Test of Computers*, 11(2):22–32, 1994.
- [74] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalijs. The VLSI-programming language Tangram and its Translation into Handshake Circuits. In *Proceedings of the European Conference on Design Automation.*, pages 384–389. IEEE, 1991.

- [75] Zhao Wang, Xiao He, and Carl M. Sechen. TonyChopper: A desynchronization ackage. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 446–453, 2014.
- [76] Stephen Williams and the Icarus Verilog Team. Icarus Verilog. <http://iverilog.icarus.com/>, 2023. Version 11.0.
- [77] Ted E Williams, Mark Horowitz, RL Alverson, and TS Yang. A Self-Timed Chip for Division. In *Stanford Conference on Advanced Research in VLSI*, pages 75–96, 1987.
- [78] Hui Wu, Weijia Chen, Zhe Su, Shaojun Wei, Anping He, and Hong Chen. A Method to Transform Synchronous Pipeline Circuits to Bundled-Data Asynchronous Circuits using Commercial EDA Tools. In *2019 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pages 1–2, 2019.
- [79] Steffen Zeidler. *Enabling Functional Tests of Asynchronous Circuits using a Test Processor Solution*. PhD thesis, BTU Cottbus-Senftenberg, 2013.
- [80] Steffen Zeidler and Miloš Krstić. A Survey about Testing Asynchronous Circuits. In *2015 European Conference on Circuit Theory and Design (ECCTD)*, pages 1–4. IEEE, 2015.
- [81] Yang Zhang, Huimei Cheng, Dake Chen, Huayu Fu, Shikhanshu Agarwal, Mark Lin, and Beerel Peter. Challenges in Building an Open-Source Flow from RTL to Bundled-Data Design. In *2018 24th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 26–27, 2018.

APPENDICES

Appendix A: Test Setup for *Ring2* in Chapter 7

A.1 ACTSIM Simulation Script for *Ring2*

```

1 echo "BEGIN SIMULATION"
2 #
3 watch scan_Ah[0] scan_Ah[1]
4 watch scan_Bh[0] scan_Bh[1]
5 watch scan_AmyW[0][0] scan_AmyW[0][1] scan_AmyW[0][2]
6 watch scan_AmyW[1][0] scan_AmyW[1][1] scan_AmyW[1][2]
7 watch scan_en_data
8 watch scan_en_control[0] scan_en_control[1]
9 watch start go[0] go[1]
10 watch J[0].fire J[1].fire
11 watch J[0].ready2go J[1].ready2go
12 watch L[0].A_me L[1].A_me
13 watch L[0].B_me L[1].B_me
14 watch L[0].B_myR[0] L[0].B_myR[1] L[0].B_myR[2]
15 watch L[1].B_myR[0] L[1].B_myR[1] L[1].B_myR[2]
16 watch L[0].req L[0].ack
17 watch L[1].req L[1].ack
18 #
19 vcd_start ring2_output.vcd
20 mode reset
21 #
22 echo "BEGIN INITIALIZE:"
23 echo "disable Joints and reset scan"
24 set start 0
25 set go[0] 0
26 set go[1] 0
27 set scan_en_data 0
28 set scan_en_control[0] 0
29 set scan_en_control[1] 0
30 cycle
31 #
32 echo "go-s now have the arbiter and J[i].fire=0, i=0..1"
33 echo "so it's OK to make start=1 from now on and to
   proceed with scan"
34 #
35 set start 1

```

```
36 set scan_en_data      1
37 set scan_en_control[0] 1
38 set scan_en_control[1] 1
39 cycle
40 #
41 echo "CONTINUE TO INITIALIZE:"
42 echo "make L[0] FULL with data [2:0]=001, L[1] EMPTY"
43 #
44 set scan_Ah[0]        0
45 set scan_Bh[0]        1
46 set scan_AmyW[0][0]   1
47 set scan_AmyW[0][1]   0
48 set scan_AmyW[0][2]   0
49 #
50 set scan_Ah[1]        0
51 set scan_Bh[1]        0
52 cycle
53 #
54 echo "done with scanning - reset scan_en_data and
      scan_en_control"
55 set scan_en_data      0
56 set scan_en_control[0] 0
57 set scan_en_control[1] 0
58 cycle
59 mode run
60 #
61 echo "END INITIALIZE"
62 echo "check that L[0].A_me=0, L[0].B_me=1, L[0].B_myR
      [2:0]=001"
63 echo "check that L[1].A_me=1, L[1].B_me=0"
64 #
65 echo "START RUN"
66 set go[0]              1
67 set go[1]              1
68 advance 200
69 echo "STOP RUN: disable J[0] after advancing 200 time
      steps and peter out"
70 set go[0]              0
71 cycle
72 echo "END RUN"
73 echo "check that L[0].A_me=0, L[0].B_me=1"
74 echo "check that L[1].A_me=1, L[1].B_me=0"
75 echo "check that data L[0].B_myR[2:0] to L[1].B_myR[2:0]"
76 echo "and data the other way shift right each action"
77 #
```

```

78 echo "END SIMULATION"
79 #
80 vcd_stop

```

A.2 Simulation Output from ACTSIM

```

1 BEGIN SIMULATION
2 BEGIN INITIALIZE:
3 disable Joints and reset scan
4 [          0] <[env]> start := 0
5 [          0] <[env]> go[0] := 0
6 [          0] <[env]> go[1] := 0
7 [          0] <[env]> scan_en_data := 0
8 [          0] <[env]> scan_en_control[0] := 0
9 [          0] <[env]> scan_en_control[1] := 0
10 [         10] <J[1].mrgo_1> J[1].fire := 0
11 [         10] <J[0].mrgo_1> J[0].fire := 0
12 go-s now have the arbiter and J[i].fire=0, i=0..1
13 so it's OK to make start=1 from now on and to proceed with
    scan
14 [         10] <[env]> start := 1
15 [         10] <[env]> scan_en_data := 1
16 [         10] <[env]> scan_en_control[0] := 1
17 [         10] <[env]> scan_en_control[1] := 1
18 CONTINUE TO INITIALIZE:
19 make L[0] FULL with data[2:0]=001, L[1] EMPTY
20 [         10] <[env]> scan_Ah[0] := 0
21 [         10] <[env]> scan_Bh[0] := 1
22 [         10] <[env]> scan_AmyW[0][0] := 1
23 [         10] <[env]> scan_AmyW[0][1] := 0
24 [         10] <[env]> scan_AmyW[0][2] := 0
25 [         10] <[env]> scan_Ah[1] := 0
26 [         10] <[env]> scan_Bh[1] := 0
27 [         20] <L[0].sFFA_req.sffHI_1> L[0].req
    := 0
28 [         20] <L[1].sFFB_ack.sffHI_1> L[1].ack
    := 0
29 [         20] <L[0].slatch_0> L[0].B_myR[0] := 1
30 [         20] <L[1].sFFA_req.sffHI_1> L[1].req
    := 0
31 [         20] <L[0].sFFB_ack.sffHI_1> L[0].ack
    := 1

```

```

32 [           20] <L[0].slatch_2> L[0].B_myR[2] := 0
33 [           20] <L[0].slatch_1> L[0].B_myR[1] := 0
34 [           30] <L[0].xorB> L[0].B_me := 1
35 [           30] <L[0].xnorA> L[0].A_me := 0
36 [           30] <L[1].xorB> L[1].B_me := 0
37 [           30] <L[1].xnorA> L[1].A_me := 1
38 [           40] <J[0].and2_1> J[0].ready2go := 1
39 [           40] <J[1].and2_1> J[1].ready2go := 0
40 done with scanning - reset scan_en_data and
    scan_en_control
41 [           40] <[env]> scan_en_data := 0
42 [           40] <[env]> scan_en_control[0] := 0
43 [           40] <[env]> scan_en_control[1] := 0
44 END INITIALIZE
45 check that L[0].A_me=0, L[0].B_me=1, L[0].B_myR[2:0]=001
46 check that L[1].A_me=1, L[1].B_me=0
47 START RUN
48 [           40] <[env]> go[0] := 1
49 [           40] <[env]> go[1] := 1
50 [           50] <J[0].mrgo_1> J[0].fire := 1
51 [           60] <L[0].sFFB_ack.sffHI_1> L[0].ack
    := 0
52 [           60] <L[1].slatch_2> L[1].B_myR[2] := 0
53 [           60] <L[1].sFFA_req.sffHI_1> L[1].req
    := 1
54 [           60] <L[1].slatch_0> L[1].B_myR[0] := 0
55 [           60] <L[1].slatch_1> L[1].B_myR[1] := 1
56 [           70] <L[1].xorB> L[1].B_me := 1
57 [           70] <L[1].xnorA> L[1].A_me := 0
58 [           70] <L[0].xnorA> L[0].A_me := 1
59 [           70] <L[0].xorB> L[0].B_me := 0
60 [           80] <J[0].and2_1> J[0].ready2go := 0
61 [           80] <J[1].and2_1> J[1].ready2go := 1
62 [           90] <J[0].mrgo_1> J[0].fire := 0
63 [           90] <J[1].mrgo_1> J[1].fire := 1
64 [          100] <L[1].sFFB_ack.sffHI_1> L[1].ack
    := 1
65 [          100] <L[0].slatch_2> L[0].B_myR[2] := 1
66 [          100] <L[0].slatch_0> L[0].B_myR[0] := 0
67 [          100] <L[0].sFFA_req.sffHI_1> L[0].req
    := 1
68 [          110] <L[1].xorB> L[1].B_me := 0
69 [          110] <L[0].xorB> L[0].B_me := 1
70 [          110] <L[0].xnorA> L[0].A_me := 0
71 [          110] <L[1].xnorA> L[1].A_me := 1

```

```

72 [      120] <J[1].and2_1>  J[1].ready2go := 0
73 [      120] <J[0].and2_1>  J[0].ready2go := 1
74 [      130] <J[1].mrgo_1>  J[1].fire := 0
75 [      130] <J[0].mrgo_1>  J[0].fire := 1
76 [      140] <L[1].slatch_1> L[1].B_myR[1] := 0
77 [      140] <L[1].sFFA_req.sffHI_1> L[1].req
    := 0
78 [      140] <L[1].slatch_0> L[1].B_myR[0] := 1
79 [      140] <L[0].sFFB_ack.sffHI_1> L[0].ack
    := 1
80 [      150] <L[0].xorB> L[0].B_me := 0
81 [      150] <L[0].xnorA> L[0].A_me := 1
82 [      150] <L[1].xnorA> L[1].A_me := 0
83 [      150] <L[1].xorB> L[1].B_me := 1
84 [      160] <J[0].and2_1>  J[0].ready2go := 0
85 [      160] <J[1].and2_1>  J[1].ready2go := 1
86 [      170] <J[0].mrgo_1>  J[0].fire := 0
87 [      170] <J[1].mrgo_1>  J[1].fire := 1
88 [      180] <L[1].sFFB_ack.sffHI_1> L[1].ack
    := 0
89 [      180] <L[0].slatch_2> L[0].B_myR[2] := 0
90 [      180] <L[0].sFFA_req.sffHI_1> L[0].req
    := 0
91 [      180] <L[0].slatch_1> L[0].B_myR[1] := 1
92 [      190] <L[0].xorB> L[0].B_me := 1
93 [      190] <L[0].xnorA> L[0].A_me := 0
94 [      190] <L[1].xnorA> L[1].A_me := 1
95 [      190] <L[1].xorB> L[1].B_me := 0
96 [      200] <J[1].and2_1>  J[1].ready2go := 0
97 [      200] <J[0].and2_1>  J[0].ready2go := 1
98 [      210] <J[1].mrgo_1>  J[1].fire := 0
99 [      210] <J[0].mrgo_1>  J[0].fire := 1
100 [      220] <L[1].slatch_2> L[1].B_myR[2] := 1
101 [      220] <L[1].sFFA_req.sffHI_1> L[1].req
    := 1
102 [      220] <L[1].slatch_0> L[1].B_myR[0] := 0
103 [      220] <L[0].sFFB_ack.sffHI_1> L[0].ack
    := 0
104 [      230] <L[0].xorB> L[0].B_me := 0
105 [      230] <L[0].xnorA> L[0].A_me := 1
106 [      230] <L[1].xnorA> L[1].A_me := 0
107 [      230] <L[1].xorB> L[1].B_me := 1
108 [      240] <J[0].and2_1>  J[0].ready2go := 0
109 [      240] <J[1].and2_1>  J[1].ready2go := 1

```

```

110 STOP RUN: disable J[0] after advancing 200 time steps and
    peter out
111 [          240] <[env]> go[0] := 0
112 [          250] <J[0].mrgo_1> J[0].fire := 0
113 [          250] <J[1].mrgo_1> J[1].fire := 1
114 [          260] <L[1].sFFB_ack.sffHI_1> L[1].ack
    := 1
115 [          260] <L[0].slatch_1> L[0].B_myR[1] := 0
116 [          260] <L[0].slatch_0> L[0].B_myR[0] := 1
117 [          260] <L[0].sFFA_req.sffHI_1> L[0].req
    := 1
118 [          270] <L[1].xorB> L[1].B_me := 0
119 [          270] <L[0].xorB> L[0].B_me := 1
120 [          270] <L[0].xnorA> L[0].A_me := 0
121 [          270] <L[1].xnorA> L[1].A_me := 1
122 [          280] <J[1].and2_1> J[1].ready2go := 0
123 [          280] <J[0].and2_1> J[0].ready2go := 1
124 [          290] <J[1].mrgo_1> J[1].fire := 0
125 END RUN
126 check that L[0].A_me=0, L[0].B_me=1
127 check that L[1].A_me=1, L[1].B_me=0
128 check that data L[0].B_myR[2:0] to L[1].B_myR[2:0]
129 and data the other way shift right each action
130 END SIMULATION

```

Appendix B: Waveforms for Mixed Implementation *Ring6*.

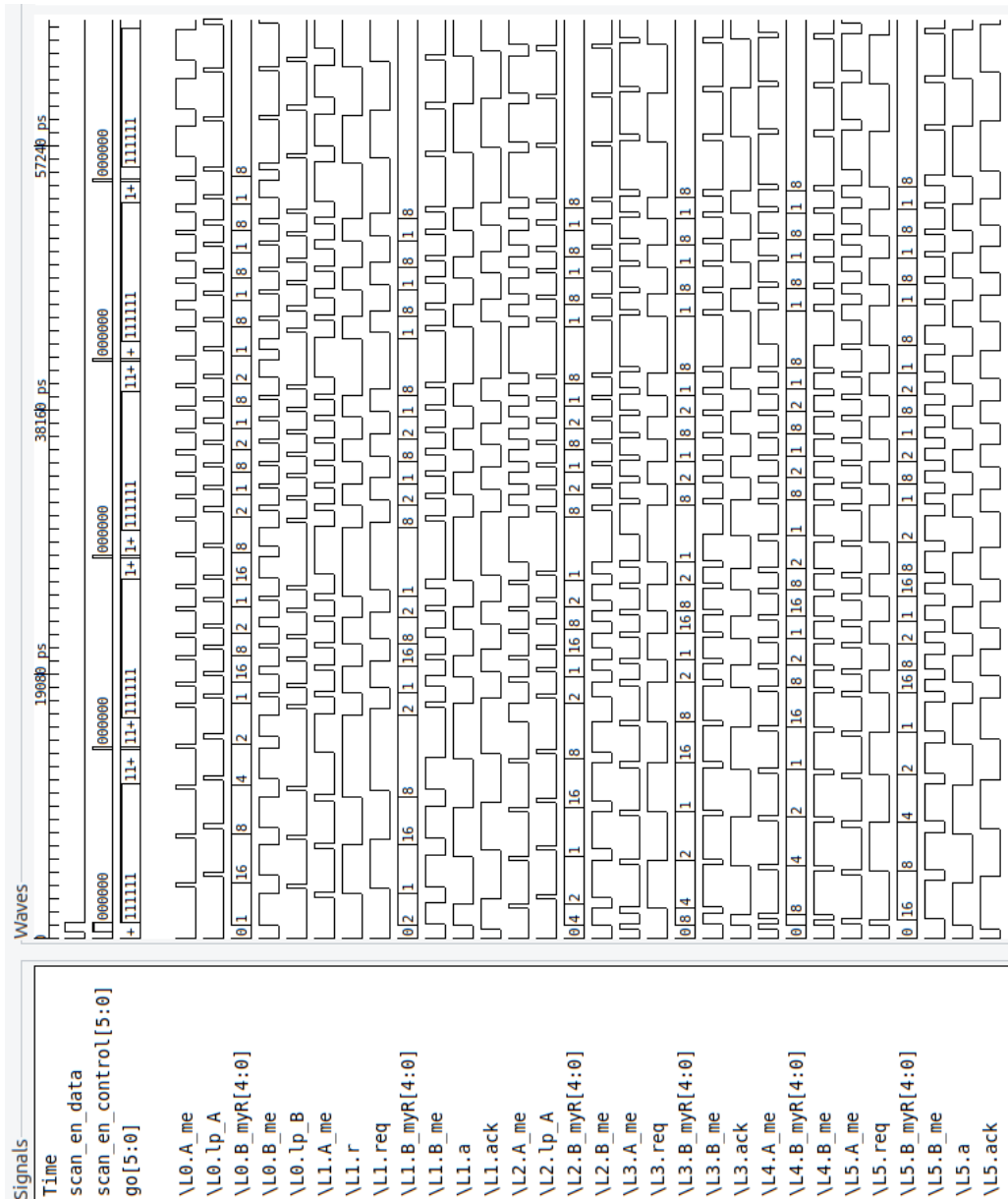


Figure B.1: Waveforms for Simulations of Mixed Implementation Ring FIFO, Ring6. They support the functionality of the design in Figure 8.1 and the Canopy graph in Figure 8.2.