IMPLEMENTATION AND EVALUATION OF ASYNCHRONOUS ARBITRATION ON FPGA FOR PHOTON COUNTING

By

Christopher J. Barr

Bachelor of Science – Computer Engineering
University of Nevada, Las Vegas
2020

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Engineering – Electrical Engineering

Department of Electrical and Computer Engineering
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2026

Copyright by Christopher J. Barr, 2026
All Rights Reserved

Abstract

Photon counting is used in many scientific applications such as medical imaging, LiDAR, and quantum optics. However, a single photon detector has a limited count rate. Under high light intensity, the photodetector sensor reaches a saturation point where incoming photons arrive faster than the device can recover from its dead time – causing counts to be missed. A solution to this is to run several detectors in parallel and merge their outputs. However, if two detectors fire at the same time, the downstream channel can only handle one at a time. Deciding which event goes first is critical. That decision is known as arbitration, and the policy used determines which channel is served first, and which one waits.

Asynchronous arbitration designs have been applied to photon counting in ASICs before. The designer has significant control over routing paths, layout, and delays on an ASIC based design. There remains a significant research gap regarding how these asynchronous topologies map to FPGA implementations. In this paper, the use of asynchronous Fixed-Priority Arbiter (FPA) and Round-Robin Arbiter (RRA) designs are chosen as solutions to compare. Initial tests will run internal and function generated signals as known number of signals. Then testing with an actual photon counting detector where a signal splitter is used to connect to a commercial photon counter and the FPGA simultaneously to determine the accuracy of the FPGA based system.

Acknowledgements

First I would like to thank God for giving me the opportunity to live life the way that I am able to today. I would also like to thank my Mom (Sang Barr), Dad (David Barr), and Sister (Alicia Barr) for pushing me and putting me in position to succeed in life. Your love and endless support will always be my motivation to keep pushing to be better.

I would like to thank Dr. Baker and Dr. Harris for not just teaching, but inspiring and never giving up on their students. Under their guidance, what originally began as a routine step towards a career transformed into a pursuit of genuine interest. I would also like to thank Angsuman Roy for his friendship and always helping me dive deeper into what interests me within electrical and computer engineering. Finally, my time in Dr. Baker's lab was made truly memorable by the friends I made there: David Santiago, Francisco Mata Carlos, James Skelly, Daniel Senda, Jazmine Boloor, Bryan Kerstetter, Shadden Kerstetter, and Dr. Sachin Namboodiri.

Beyond the lab and classroom, I want to express my deepest love and gratitude to my partner, Nancy Quach. Thank you for always supporting me during those sleepless nights, and supporting my ambitions.

Contents

Abstract iii

Acknowledgements iv

List of Tables vii

List of Figures ix

List of Abbreviations xii

1 Introduction 1
- 1.1 Photon Counting and Detector Dead Time . . . 1
- 1.2 Parallel Detector Arrays and the Shared-Readout Bottleneck . . . 1
- 1.3 Asynchronous Arbitration in FPGA Fabric . . . 2

2 Background and Related Work 3
- 2.1 Single-Photon Avalanche Diodes and Dead Time . . . 3
- 2.2 Self-Timed Design and Four-Phase Handshaking . . . 4
- 2.3 The Muller C-Element . . . 5
- 2.4 Arbitration Policies: Fixed Priority and Round Robin . . . 5
- 2.5 Arbiter Tree Structure . . . 7
- 2.6 ASIC Implementations . . . 8
- 2.7 Gap Addressed by this Work . . . 9

3 System Architecture 10
- 3.1 FPGA Platform: Digilent Eclypse Z7 . . . 11
- 3.2 Asynchronous Event Capture . . . 12
- 3.3 Grants Per-Channel Counting . . . 13
- 3.4 Clock-Domain Crossing (CDC) and AXI-Lite Readout . . . 14
- 3.5 Single-Source Build Strategy . . . 16
- 3.6 Design Matrix . . . 17

4 Arbiter Implementation in FPGA Fabric 18
- 4.1 Mapping Muller Gates to Look-Up Tables . . . 18
- 4.2 The No Arbiter Module . . . 19
- 4.3 The 2-to-1 Fixed-Priority Arbiter Cell . . . 20
- 4.4 The 2-to-1 Round-Robin Arbiter Cell . . . 22
- 4.5 Fixed-Priority Arbiter Tree . . . 23
- 4.6 Round-Robin Arbiter Tree . . . 24
- 4.7 Delay Chains as Tuning Parameters . . . 25
- 4.8 Constraining Intentional Combinational Loops . . . 25
- 4.9 Timing Closure Across the Design Matrix . . . 26

5 Behavioral Verification **27**
5.1 Unit-Cell Testbench . . . 28
5.2 Tree-Level Testbench . . . 32
5.3 Limits of Behavioral Simulation for Placement-Dependent Logic . . . 34

6 Experimental Methodology **35**
6.1 In-Fabric Signal Generator . . . 36
6.2 External Pulse Generators . . . 37
6.3 SPAD Detector . . . 39

7 Results **46**
7.1 Resource Utilization . . . 46
7.2 In-Fabric Signal Generator . . . 48
7.2.1 Two-Channel: No Arbiter vs FPA Cell vs RRA Cell . . . 49
7.2.2 Four-Channel: No Arbiter vs FPA Tree vs RRA Tree . . . 52
7.3 External Pulse Generators . . . 61
7.3.1 Two-Channel: FPA Cell vs RRA Cell . . . 62
7.3.2 Four-Channel: FPA Tree vs RRA Tree . . . 66
7.4 SPAD Detector . . . 70
7.4.1 Two-Channel: FPA Cell vs RRA Cell . . . 71
7.4.2 Four-Channel: FPA Tree vs RRA Tree . . . 73

8 Conclusions and Future Work **75**
8.1 Conclusion . . . 75
8.2 Future Work . . . 75

A Verilog Source Code **77**
A.1 Asynchronous Flip-Flop . . . 77
A.2 No Arbiter . . . 77
A.3 FPA Cell (2×1) . . . 78
A.4 RRA Cell (2×1) . . . 79
A.5 FPA Tree (4×1) . . . 81
A.6 RRA Tree (4×1) . . . 82
A.7 Asynchronous Counter . . . 83
A.8 Synchronous Counter . . . 85
A.9 Complete Source Code Repository . . . 87

B Complete Edge-Case Counts **88**

Bibliography **94**

C Curriculum Vitae **96**

List of Tables

3.1 AXI-Lite register map 16
3.2 Design Matrix 17

4.1 Post-implementation timing closure per build. 26

5.1 Behavioral verification of the FPA Cell (2×1) and RRA Cell (2×1). The ✓ means the test passed in behavioral simulation, and the — means it is not applicable. 28
5.2 Behavioral verification of the FPA Tree (4×1) and RRA Tree (4×1). The ✓ indicates the test passed in behavioral simulation, and the — indicates it is not applicable. 32

6.1 DG535 pulse delay generator output delay settings. 38

7.1 LUT utilization by design. 46
7.2 Propagation delay through each designs channel 0. 47
7.3 Estimated on-chip power by build. 48
7.4 Two-channel (2×1) edge-case 1 million events counted per active channel. Sorted by active channels and phase patterns. 'A' is the first channel (ch0 for No Arbiter/FPA Cell, ch2 for RRA), and 'B' is the second channel (ch1 for No Arbiter/FPA Cell, ch3 for RRA). The phase patterns are offset by one clock (10 ns). Shaded cells have both channels arriving at the same clock cycle. 50
7.5 RRA Tree (4×1) failed edge cases (period 30 ns, 1M events per channel. Shaded cell means a difference of more than 1% of what was offered). (see complete counts in Appendix B). 54
7.6 FPA Tree (4×1) failed edge cases (period **20 ns**, 1M events per channel. Shaded cell means a difference of more than 1% of what was offered). (see complete counts in Appendix B). 58
7.7 RRA Tree (4×1) failed edge cases (period 20 ns, 1M events per channel. Shaded cell means a difference of more than 1% of what was offered). (see complete counts in Appendix B). 60
7.8 FPA (2×1) Cell edge-offset sweep: events lost of 600,000 per channel. 62
7.9 RRA (2×1) Cell edge-offset sweep: events lost of 600,000 per channel. 63
7.10 FPA (2×1) Cell: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%. 64
7.11 RRA (2×1) Cell: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%. 65
7.12 FPA (4×1) Tree edge-offset sweep: events lost of 600,000 per channel. Negative values indicate more events counted than given. 66
7.13 RRA (4×1) Tree edge-offset sweep: events lost of 600,000 per channel. . . . 67
7.14 FPA (4×1) Tree: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%. 68

7.15 RRA (4×1) Tree: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%. 69

B.1 No Arbiter (4×1) total counts in edge cases (period **30 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle. 88
B.2 FPA Tree (4×1) total counts in edge cases (period **30 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle. 89
B.3 RRA Tree (4×1) total counts in edge cases (period **30 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle. 90
B.4 No Arbiter (4×1) total counts in edge cases (period **20 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle. 91
B.5 FPA Tree (4×1) total counts in edge cases (period **20 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle. 92
B.6 RRA Tree (4×1) total counts in edge cases (period **20 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle. 93

List of Figures

2.1 (a) Bundled-data (push) channel and (b) four-phase handshake protocol [15]. 4
2.2 2-to-1 FPA cell schematic [16] 6
2.3 16-to-1 FPA tree structure example [1] 8

3.1 System architecture block diagram 10
3.2 Top view of the Eclypse Z7 [6] 11
3.3 Asynchronous Flip Flop to take in signal pulses 12
3.4 Asynchronous Counter Block Diagram 13
3.5 Asynchronous to synchronous counter block diagram 14
3.6 Logic hazard from combinational logic [8] 16

4.1 Muller C-element [13] as implemented within the FPA and RRA cells. Resetn is active LOW. 18
4.2 No Arbiter Block Diagram 19
4.3 FPA Cell (2×1) Block Diagram 20
4.4 RRA Cell (2×1) Block Diagram 22
4.5 FPA Tree (4×1) Block Diagram 23
4.6 RRA Tree (4×1) Block Diagram 24

5.1 FPA Cell (2×1) behavioral simulation of tests 2 and 3. 29
5.2 RRA Cell (2×1) behavioral simulation of Tests 2 and 3. 30
5.3 Both FPA and RRA Cells (2×1) behavioral simulation of Test 6. Both arbiters resolve the simultaneous arrival successfully. 30
5.4 Both FPA and RRA Cells (2×1) behavioral simulation of Tests 15 and 16. . 31
5.5 Both FPA and RRA Trees (4×1) behavioral simulation of Test 14 and 15. . 33

6.1 Full workstation setup. 35
6.2 Signal generator behavior testbench results. 36
6.3 Offset testing using Stanford Research Systems delay pulse generator. 37
6.4 Single-channel throughput testing using the Agilent 33250A function generator. 38
6.5 Photon counting test setup overview. 39
6.6 Current driven LED. 40
6.7 Integrating sphere for light optics attenuation. 41
6.8 The Count-100C SPAD. 42
6.9 SR400 and SR430 photon counters. 43
6.10 Eclypse Z7 wired up for photon testing. 43
6.11 COUNT-100C SPAD output pulses captured on an oscilloscope 44
6.12 Software Interface (Serial Connected) 44
6.13 Function generators used for 30 MHz contention request signals. Agilent is not depicted in image accurately for this test. 45

7.1 Two-channel throughput efficiency by active-channel mask and phase pattern for the No Arbiter, FPA Cell, and RRA Cell designs (period = **30 ns**, 1 million events per channel) 49
7.2 Two-channel throughput efficiency by active-channel mask and phase pattern for the No Arbiter, FPA Cell, and RRA Cell designs (period = **20 ns**, 1 million events per channel) 50
7.3 Two-channel throughput efficiency versus input event rate per channel. The dashed line marks the COUNT-100C dead time (23.8 MEvents/s). 51
7.4 Four-channel throughput efficiency, No Arbiter (4×1) (period = **30 ns**, 1 million events per channel). 52
7.5 Four-channel throughput efficiency, FPA Tree (4×1) (period = **30 ns**, 1M events per channel). 53
7.6 Four-channel throughput efficiency, RRA Tree (4×1) (period = **30 ns**, 1M events per channel). 54
7.7 Four-channel throughput efficiency, No Arbiter (4×1) (period = **20 ns**, 1 million events per channel). 56
7.8 Four-channel throughput efficiency, FPA Tree (4×1) (period = **20 ns**, 1 million events per channel). 57
7.9 Four-channel throughput efficiency, RRA Tree (4×1) (period = **20 ns**, 1 million events per channel). 59
7.10 Four-channel throughput efficiency versus input event rate per channel. The dashed line marks the COUNT-100C dead-time limit (23.8 MEvents/s). . . . 61
7.11 FPA (2x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator 62
7.12 RRA (2x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator 63
7.13 FPA (2×1) Cell: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events 64
7.14 RRA (2×1) Cell: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events 65
7.15 FPA (4x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator 66
7.16 RRA (4x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator 67
7.17 FPA (4×1) Tree: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events 68
7.18 RRA (4×1) Tree: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events 69
7.19 FPA (2×1) Cell: ch0 SPAD count rate (cps) vs. light source drive current (µA), SR400 and FPGA comparison, ch1 at 30 MHz by the function generator. 71
7.20 RRA (2×1) Cell: ch2 SPAD count rate (cps) vs. light source drive current (µA), SR400 and FPGA comparison, ch3 at 30 MHz by the function generator. 72
7.21 FPA (4×1) Tree: ch0 SPAD count rate (cps) vs. light source drive current (µA), SR430 and FPGA comparison, ch1/ch2/ch3 at 30 MHz by the function generators. 73

7.22 RRA (4×1) Tree: ch0 SPAD count rate (cps) vs. light source drive current (µA), SR430 and FPGA comparison, ch1/ch2/ch3 at 30 MHz by the function generators. 74

List of Abbreviations

APD	Avalanche Photodiode
ASIC	Application-Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CDC	Clock-Domain Crossing
DRC	Design Rule Check
FPA	Fixed-Priority Arbiter
FPGA	Field-Programmable Gate Array
LED	Light-Emitting Diode
LiDAR	Light Detection and Ranging
LUT	Look-Up Table
MUX	Multiplexer
PL	Programmable Logic
PMOD	Peripheral Module
PS	Processing System
RRA	Round-Robin Arbiter
RTL	Register-Transfer Level
SPAD	Single-Photon Avalanche Diode
STA	Static Timing Analysis
TTL	Transistor-Transistor Logic
WHS	Worst Hold Slack
WNS	Worst Negative Slack

Chapter 1: Introduction

Photon counting is a vital technique in numerous scientific applications such as fluorescence lifetime imaging, medical imaging, LiDAR, and quantum optics. It is used as a method for counting individual particles of light one by one, rather than measuring the total light intensity as a photocurrent. The sensitivity of a photon counter is what makes the tool so important. It allows for the detection of light in sensitive applications where the inherent noise of the photodetector or electronics is higher than that of the photocurrent.

1.1 Photon Counting and Detector Dead Time

Photon counting techniques only indicate the presence of a breakdown event (photon or dark count) and not the actual intensity of the event e.g. one photon or ten photons arriving at the exact same moment will create only one pulse. The count pulse typically has a finite duration due to the photodetector needing time to reset the bias voltage across it or due to the bandwidth limitations of the first stage electronics. Improving the speed of photodiodes and electronics is an active area of research, but it is not the focus of this thesis.

1.2 Parallel Detector Arrays and the Shared-Readout Bottleneck

A single SPAD has a max count rate, capped by its dead time. Many applications of SPADs require higher maximum photon count rates, or counting dynamic range. Multiple SPADs can be placed in parallel for applications where spatial resolution is not needed and only an aggregate photon count is desired. A major bottleneck is the amount of readout pins needed for all the parallel detectors. The most obvious solution would be to sum multiple detectors into a single output pin. A problem that arises from summing multiple SPADs is that if multiple breakdown events occur at the same time and overlap, the counting electronics may not be able to resolve distinct events due to signal collision. For example, if

two SPADs fire at the same time the readout circuitry merges the two and one photon count is lost. To prevent this loss, an arbiter circuit of some sort is required. An arbiter orders colliding requests so that no signal event is lost and all events are accounted for up to the inherent limits of the arbiter or proceeding logic.

1.3 Asynchronous Arbitration in FPGA Fabric

An arbiter is capable of serializing colliding requests at the readout point. Because photon arrivals are random, a self-timed arbiter responding to request arrival responds immediately instead of waiting for the next clock edge. Prior work shows that asynchronous Fixed Priority Arbiters (FPA) were built for SPAD readouts in ASIC [16]. But ASIC is costly, and the development time can take a while. Using an FPGA, the cost drops significantly, development time shortens, and the FPGA can assist in validating designs before committing to ASIC fabrication (intermediary stage). However, FPGA tools and fabric are not built to handle asynchronous logic. This thesis investigates whether or not ASIC arbiter designs are able to retain its behavior when implemented in FPGA fabric, where its own tools control logic placement and routing rather than the designer.

Chapter 2: Background and Related Work

2.1 Single-Photon Avalanche Diodes and Dead Time

A Single-Photon Avalanche Diode (SPAD) is a p-n junction biased at voltage, V_A, above its breakdown voltage, V_B. This condition is known as Geiger mode [3]. With an electric field so high, a single charge carrier injected into the depletion layer is accelerated enough to create a new electron-hole pair through impact ionization. More carriers are created, which produces a self-sustaining avalanche. This yields a current pulse large enough to be detected. Photon detection efficiency is increased with an increase in excess bias voltage, V_E.

$$V_E = V_A - V_B \tag{2.1}$$

Inside the SPAD, thermal generation effects may produce current pulses large enough to be detected, even without any influence from external light sources. This is known as a dark count. These dark counts represent the internal noise of the SPAD. Increasing the V_E also increases the dark count rate. Once an avalanche is detected, the bias voltage V_A is lowered to quench it, allowing for another photon to be detected.

While V_A is being lowered to quench and reset, the detector is unable to respond to incoming photons arriving in this window. This is known as the dead time. As the photon arrival rate increases, the more photons fall within the dead time and are missed, saturating the count rate [2]. The maximum count rate of a SPAD can be calculated by the inverse of the dead time. It should be known that at exceedingly high light levels, the SPAD's dead time can increase leading to paralysis and actually result in a reduction in count rate with increasing light levels [3]. This area of operation should be avoided in order to realize the maximum count rate of a SPAD which is also known as the non-paralyzable count-rate.

The SPAD module used in this paper is the Laser Components COUNT-100C, which is a quenched silicon detector module, sensitive from 400 to 1000 nm [10]. The dead time for the detector is 42 ns minimum. This is roughly 23.8 MEvents/s. Internal electronics prevent falling counts after saturation and therefore its dead time is non-paralyzable, which bring the count rate closer to the rate of max detection. The COUNT-100C has a max dark count rate of 100 counts/s, with a Transistor-Transistor Logic (TTL) pulse width of 15 ns [10].

2.2 Self-Timed Design and Four-Phase Handshaking

Self-timed (asynchronous) designs have multiple advantages over clock-based (synchronous) designs such as higher operating speeds, lower power consumption, less emission of electro-magnetic noise, better composability and modularity, and no clock distribution and clock skew problems [15]. With synchronous designs, it guarantees a minimum of at least a two-sided timing constraint that uses both setup and hold time windows which revolves around the clock edge. Wire placement delays contribute additional difficulty in synchronous designs due to skew. However, asynchronous designs rely on a different form of communication than relying on the clock, and that is handshake protocols. Handshake protocols allow for two modules or registers to relay signal arrivals. These are the two wires of `request` and `acknowledge` based handshake protocols, shown in Figure 2.1.

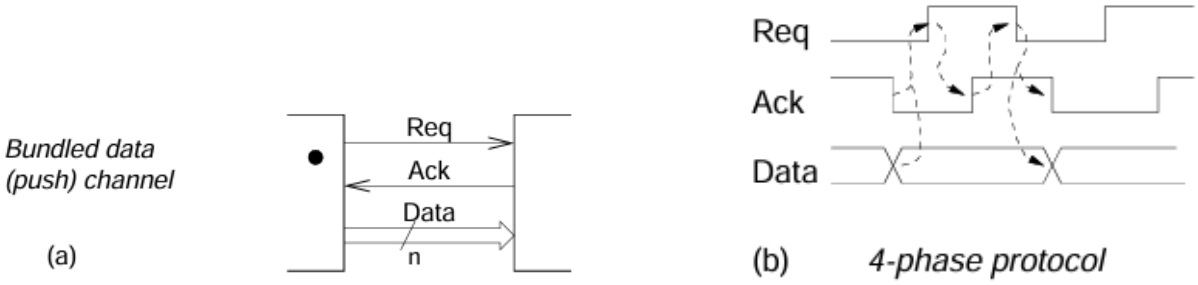


Figure 2.1: (a) Bundled-data (push) channel and (b) four-phase handshake protocol [15].

The way a 4-phase protocol works is the transmitting module sends a request, acknowledge of the receiving module is set high, the transmitting module receives the acknowl-

edge and sets the request low, and lastly the receiving module acknowledges that and is set low. That is a 4-phase protocol. For reference, this is also depicted in Figure 2.1b. The bundled data can be based on whatever information needs to be transmitted, this thesis will be using bundled data as channel origin. The key thing about the bundled data is that it needs to be validated first before the handshake proceeds. This means the bundled data needs to arrive the handshake protocol departs without it (discussed later in Section 4.7). Using the 4-phase handshake protocol, suggested by Sparsø [15], this will be the core communication technique between the arbiter and the rest of the asynchronous modules.

2.3 The Muller C-Element

Muller gates play a significant role in asynchronous circuits today. Because asynchronous circuits rely on handshakes between modules, hazards may present itself due to handshake time constraints. To prevent such hazards, a Muller gate can be implemented to hold values until a handshake arrives to ensure completeness of communication. A Muller C-element is a memory-based set of logic gates, made up of three AND gates, and one OR gate (Figure 4.1a). The design is functional hazard free because it prevents problems where `01` $\rightarrow$ `10` and `10` $\rightarrow$ `01`. Meaning, if both inputs flip states, the circuit may see `11` or `00` for a brief moment, which sets the output to `1` or `0`, respectively [13]. The other issue is the logic hazard that may be found in the FPGA module design post-synthesis, if the Muller C-element is not implemented properly. This is because if the synthesis does not place all the Muller C-elements logic into one LUT, there may be some routing delay for one input over the other [13]. The fix to this is by ensuring the Muller gate is implemented as a single LUT using LUT instantiation [13].

2.4 Arbitration Policies: Fixed Priority and Round Robin

When request signals are sent to hardware using a single channel, the receiving module should have no concern resolving them. However, when that same module has to deal with

two channels of request signals, a problem of signal contention is introduced. Without an arbiter to arbitrate the two channels simultaneous arrivals, the module will have no way to resolve it, causing the signals to overlap and merge into one. An arbiters role is to decide which channel will go first based on its priority policy. The two policies that will be covered in this section and tested throughout the paper is the Fixed Priority Arbiter (FPA) and Round Robin Arbiter (RRA).

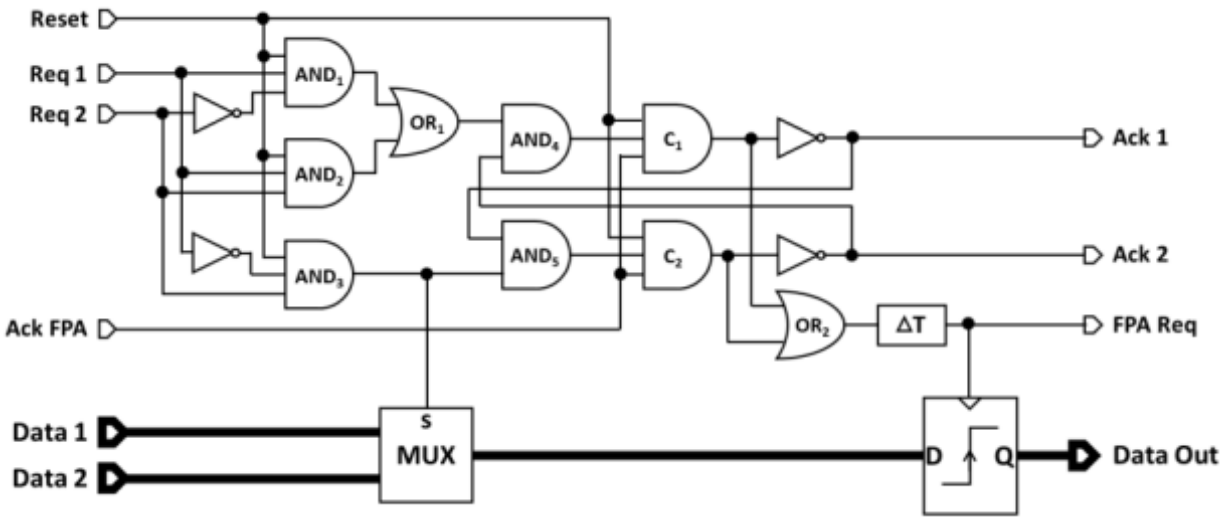


Figure 2.2: 2-to-1 FPA cell schematic [16]

In an FPA design, the priority is fixed, which makes it a minimal static design. When there is contention, the higher priority channel will always win. The strengths of the design show up in its simplicity, smaller size, high event rate, and its deterministic behavior [16]. The fundamental weakness of the design is the effects of always choosing the highest priority channel, it leads to starvation of the lowest priority channel when requests are rapidly injected [16]. The arbiter design shown in Figure 2.2 was created for an ASIC to take in SPAD detection signals. It will be used as the FPA cell design modeled throughout this paper.

In an RRA design, the priority is rotated, which makes the design more complex and dynamic. When there is contention, the higher priority channel will win, but the priority will rotate and point to the other channel. The strengths of the design show up in its fairness

and starvation prevention [12]. The weakness of the design lies in its additional LUT logic, adding more delay to the critical path [12]. The RRA design chosen for this paper is the same baseline design seen in Figure 2.2, but the priority network is changed to reflect a traditional RRA logic-based design, as seen later in Figure 4.4.

As long as the arbiter is not saturated, every channel request is eventually accounted for. But at saturation when requests arrive faster than the arbiter can serve, the priority network policy performance differences will appear.

2.5 Arbiter Tree Structure

An arbiter cell contains all the logic that takes in channel requests and data. An arbiter tree use those cells like nodes in a binary tree. This means the cells are cascaded, where the leaf cells take the channel requests, and the root cell drives the readout. The tree structure allows N channels to scale based on $\log_2 N$ levels. An example of how the arbiter tree structure looks can be seen in Figure 2.3.

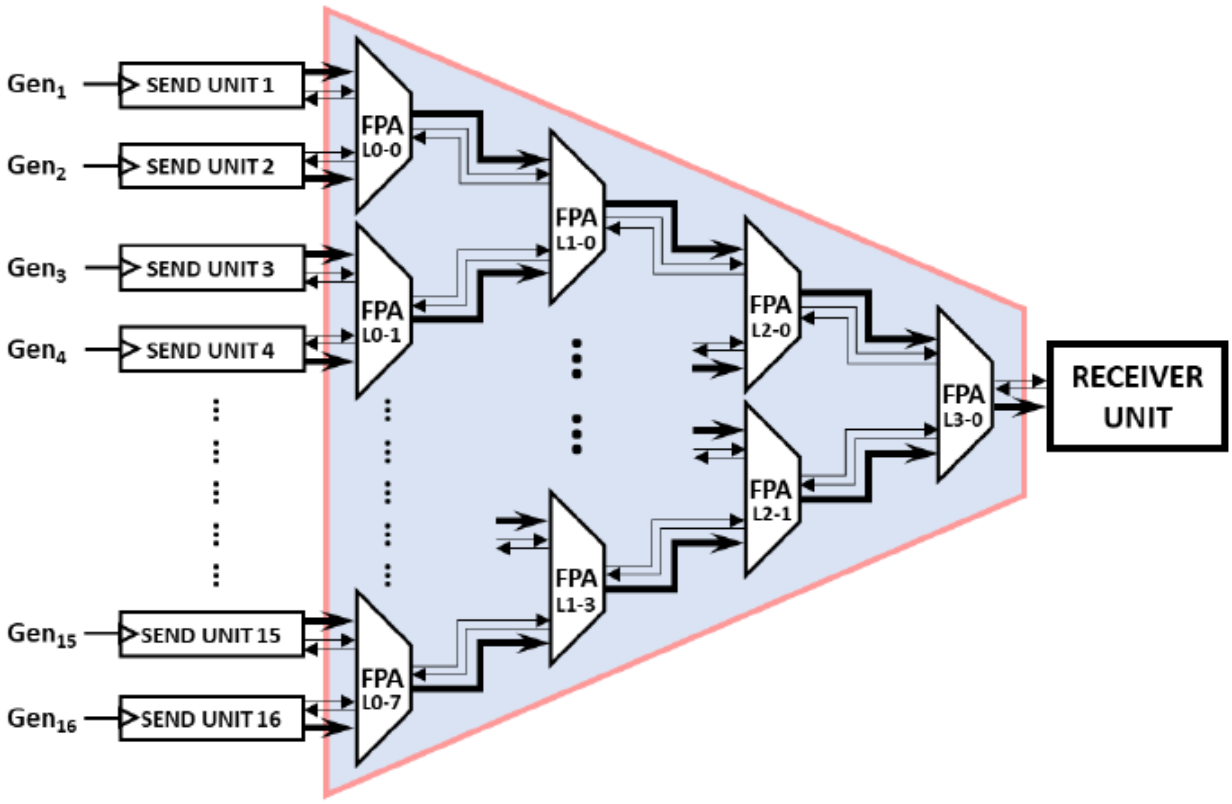


Figure 2.3: 16-to-1 FPA tree structure example [1]

Each level to the arbiter trees structure expands the data bus' origin width by one bit. And each child's data output is one of the two parents data input. The arbiter is essentially funneling the data upward till it reaches the root. Once the receiver unit (root) obtains it, an acknowledge is sent back to the child cell, which it then continues to propagate downward facing the intermediate child cells till it reaches the leaf, and then completes the full handshake to the original requester. A 4×1 arbiter tree can be seen later in Sections 4.5 and 4.6. However, for every level added to the tree, its an additional latency delay due it adding an extra handshake.

2.6 ASIC Implementations

Asynchronous designs are commonly found in ASIC designs, as it does not have to deal with placement delays like FPGA's. With ASIC designing, the designer can control the

amount of delay in a path. Asynchronous arbiter designs specifically have a long history in custom integrated circuits to maximize performance. Some early work used a mutual exclusion element as the core of an arbiter to perform four-phase handshakes using a two-way communication link [17]. The designs also relied on intentional delays between handshakes to introduce delay, preventing unstable grant outputs while the rest of the circuit settled. This technique is also performed in this paper. The choice to rely on buffers can alo be seen used with ΔT in Figure 2.2 [16].

2.7 Gap Addressed by this Work

Prior asynchronous arbiters for SPAD readout were built for ASICs, in simulation [16] and in fabricated silicon [1]. FPGA implementations are less common. It is shown that it is possible from previous related work [13] [11]. The primary constraints resides in the fact that the tools are geared towards synchronous designs. Otherwise, the software tools will try and push-back without proper constraints: combinational loops are flagged as errors, static timing analysis does not work on self-time paths, and routing delays is set by the synthesis tool. Tree's from other designs show a heavy reliance on intentional delay buffers. One open observation, why is the drain order 0, 2, 1, 3 and not 0, 1, 2, 3 for the FPA tree [16]? This paper discovered the explanation shown in Chapter 5. Otherwise, no other study has characterized asynchronous photon counting arbiters within FPGA fabric in a way that the ASIC assumptions will differ. Another gap is that FPA and RRA have not been compared under indentical conditions for a photon counting application. The cost of creating an ASIC is significant, so discovering FPGA limitations for this case may serve to be useful. This paper will go through the FPA and RRA cell/tree implementations, then evaluating the designs with three different test methodologies: in-fabric event generator, function generator, and a real SPAD.

Chapter 3: System Architecture

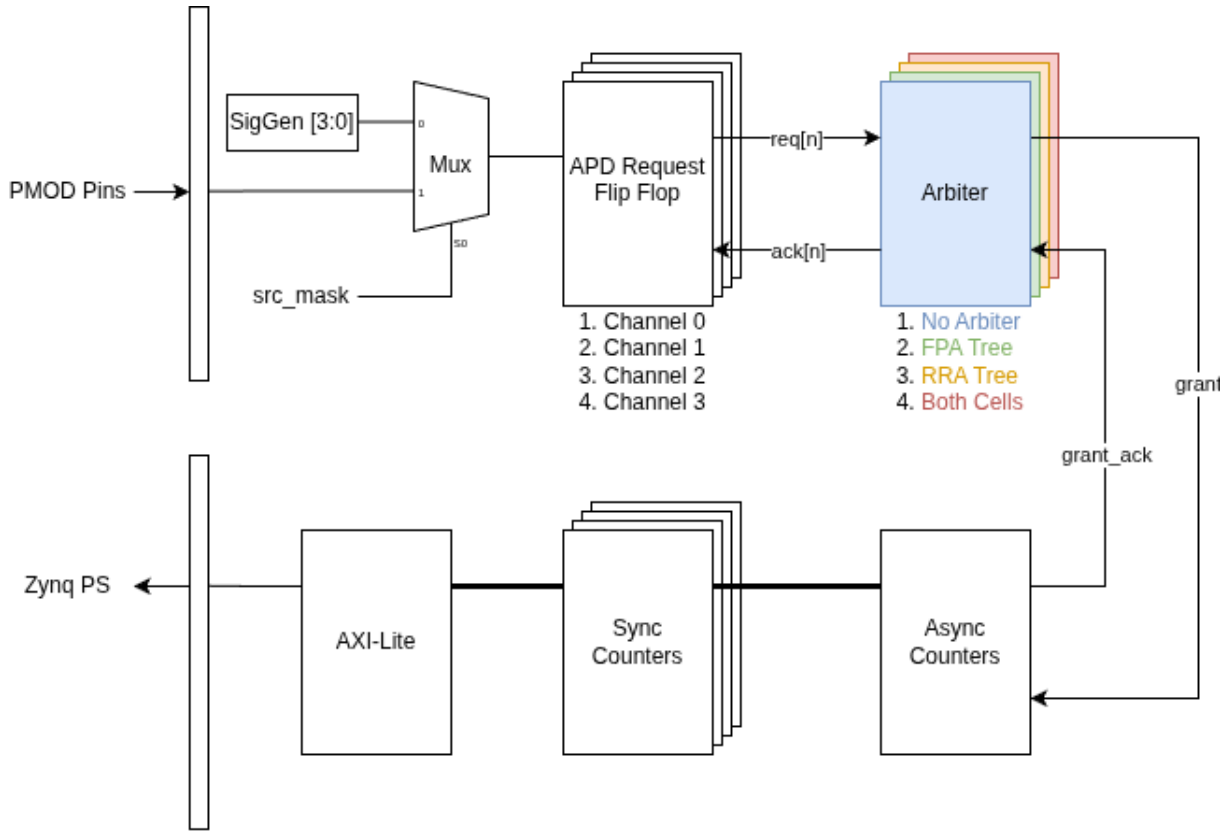


Figure 3.1: System architecture block diagram

The overview of the system's architecture only contains the primary signal path, as seen in Figure 3.1. Hidden from this figure are the resets, clock, wires for the SigGen (Signal Generator) from the AXI Lite, and the parameters. This is not meant to serve as a full datapath block diagram, but rather to represent the signal's path through the system. To briefly go through it, the input signal to the system is selected by a source mask choosing between the in-fabric signal generator and an external signal from the PMOD pins. The source is masked to choose a mix between four channels: two internal generated signals and two external signals, or three and one, etc. Next, each signal is sent to its own flip-flop and ties req[n] to the value '1' until ack[n] from the Arbiter block handshakes to clear it; where 'n'

is the channel number. The arbiter then takes the multiple signals and arbitrates them like a funnel, from two to one or four to one. The single outputted signal is sent out as a grant to be counted asynchronously and is only cleared to count the next one once grant_ack is returned. Then, from the asynchronous counter, the synchronous counters carry those values counted through a Clock Domain Crossing (CDC) process. Afterwards, the counts are sent through the AXI-Lite to the Processing System (PS) for the register values to be viewed via serial connection.

3.1 FPGA Platform: Digilent Eclypse Z7



Figure 3.2: Top view of the Eclypse Z7 [6]

The Digilent Eclypse Z7 board was chosen because its ARM processor cores (PS) and 7-series Programmable Logic (PL) share the same die, connected through AXI [4]. The three primary ports and switch used on this board for this project were the PMOD pins for the "SPAD Inputs" (as shown in Figure 3.1), serial connection port to view the register values as portrayed by the PS, and the power switch. Because the PS and PL communicate over AXI, test control and readout run on the board itself, with no separate host interface. The 7-series PL is built from 6-input LUTs, a widely used architecture, so a full Muller C-element fits in a single LUT. The board also holds SYZYGY ports which provide higher

speed throughput bandwidth that may be used in future work [4]. This board also serves as a good representative of a common FPGA hardware for a moderate price.

3.2 Asynchronous Event Capture

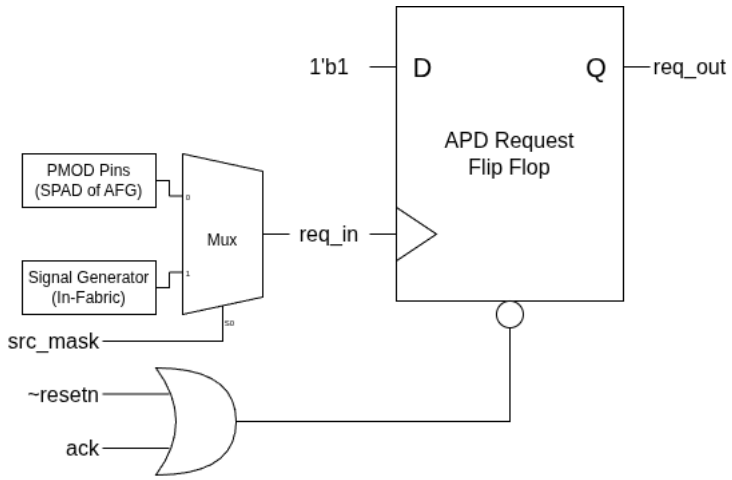


Figure 3.3: Asynchronous Flip Flop to take in signal pulses

The Avalanche Photodiode (APD) request flip-flop, shown in Figure 3.3, captures each incoming event requests and holds it until the arbiter acknowledges it. Its input comes from either external signals via PMOD pins or internal signals via the signal generator. Its output is the channel's request to the arbiter. A rising edge on `req_in` causes an event trigger to set `req_out` high, since port D is tied to 1'b1. The inverted `resetn` and `ack` signals are ORed together to create a clear to drive the `req_out` low, which completes each channel's side of the handshake. The edge-triggered flip-flop is used to capture an event regardless of the event's pulse width. There are four APD request flip-flops instantiated to hold one event per channel. From here, it then gets passed to the arbiter to perform arbitration between the four or two channels.

3.3 Grants Per-Channel Counting

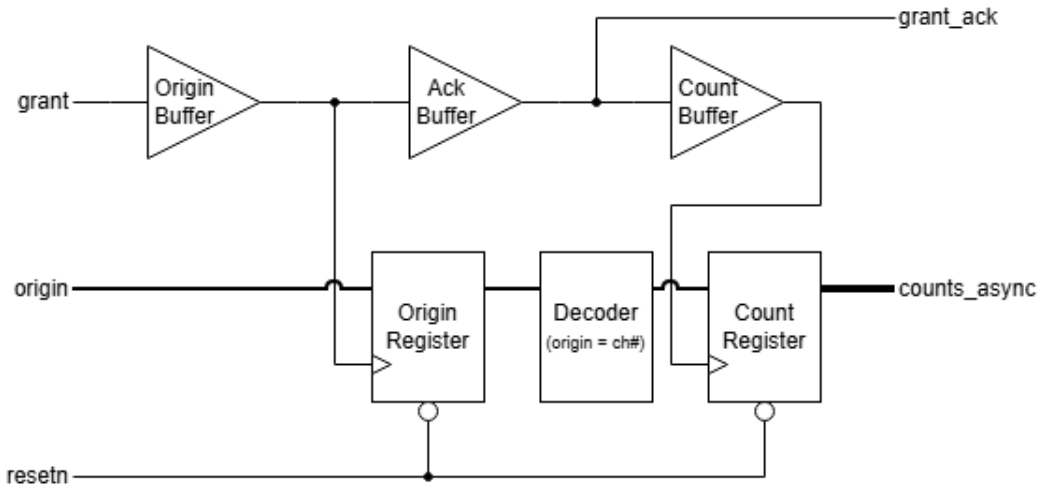


Figure 3.4: Asynchronous Counter Block Diagram

Once arbitration is completed by the arbiter, its outputs are taken in by the asynchronous counter. The task of the asynchronous counter is to count every grant that arrives and send it out in the proper channel, as shown in 3.4. The inputs are the `grant`, `origin`, and `resetn`. The `grant` represents the signal event of the winning channel, the `origin` identifies the winning channel, and `resetn` is the combined hardware and software reset. The outputs are `grant_ack` and `counts_async`. The `grant_ack` is the second or fourth part of the four-phase handshake allowing the arbiter to clear its Muller C-element and ready the next event. And the `counts_async` is a 128-bit counter output (or 64-bit for cells), 32-bits per channel. When a `grant` arrives, it passes through an origin buffer to ensure the `origin` signal arrives to the origin register on time. Then, the already delayed `grant` is taken through an ack buffer as a means to control the arbiter's ack signals timing. Finally, the twice delayed `grant` is buffered one last time at the counter buffer to edge-trigger the count register to increment the winning channel's register based on the decoder. The decoder and count register blocks may be deceiving in 3.4, so its process is shown below in Verilog HDL code.

```
// Async Counters, where "cnt_clk" is the output of "Count Buffer"
always @(posedge cnt_clk or negedge resetn) begin
    if (!resetn) begin
        cnt[0] <= 32'd0;
        cnt[1] <= 32'd0;
        cnt[2] <= 32'd0;
        cnt[3] <= 32'd0;
    end else begin
        if (origin_r == 2'b00)
            cnt[0] <= cnt[0] + 32'b1;
        if (origin_r == 2'b01)
            cnt[1] <= cnt[1] + 32'b1;
        if (origin_r == 2'b10)
            cnt[2] <= cnt[2] + 32'b1;
        if (origin_r == 2'b11)
            cnt[3] <= cnt[3] + 32'b1;
    end
end
```

To receive all grants and have them register to the correct channel, delays are required. Without the three delays, it could create issues ranging from missing a count due to edge trigger speed, counting an event to the wrong register, or events arriving too fast from the arbiter. These are three of the four timing knobs within the system design. The fourth is the ΔT buffer located within the FPA and RRA cells.

3.4 Clock-Domain Crossing (CDC) and AXI-Lite Readout

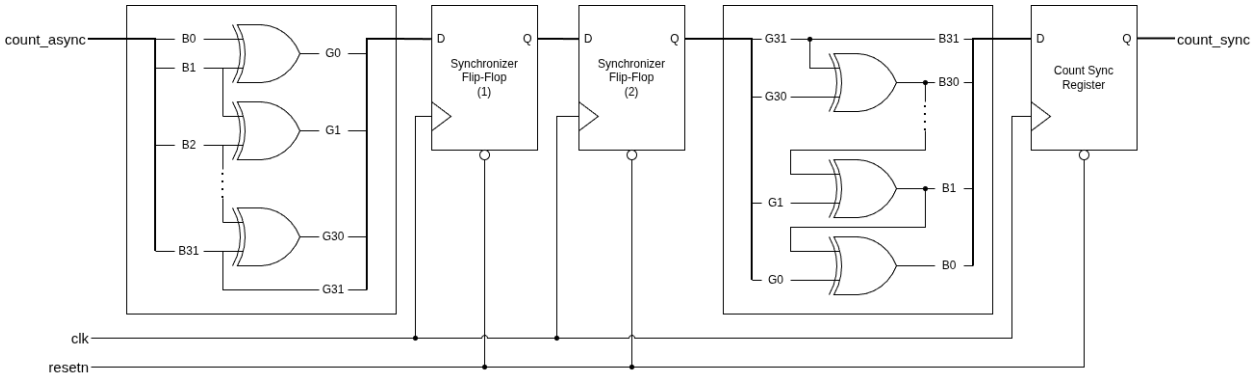


Figure 3.5: Asynchronous to synchronous counter block diagram

As the asynchronous counter is counting, the synchronous counters continuously sample its values with a 100 MHz system clock. They are then held in registers readily available for the software to read over the AXI-Lite. The synchronous counter is instantiated four times to sample all four channel counts. Its inputs are `count_async`, `clk`, and `resetn`. The `count_async` is a 32-bit per-channel input that represents the specific channels total count. The `clk` is the systems 100 MHz clock, and the `resetn` is the combined hardware and software reset. The output `count_sync` is the 32-bit per-channel register that the software will read over from the AXI-Lite. When the `count_async` is being sampled, it must CDC over from the asynchronous domain to the 100 MHz domain. In order to do that, it goes through a Binary-to-Gray code (B2G) conversion, two synchronizer flip-flops, a Gray code to Binary (G2B) conversion, and finally the count sync register, as shown in Figure 3.5. The `count_async` goes through a B2G process and back to safely change 1-bit per increment. Afterwards, it goes through two synchronizer flip-flops for metastability and to synchronize with the 100 MHz clock domain on the synchronous side. The results are then reverted back from Gray code to binary and gets stored in a register edge-triggered by the systems 100 MHz clock.

It was later discovered that although Figure 3.5 correctly CDC the asynchronous counter to synchronous, the design does not account for hazards. The binary-to-Gray conversion is combinational and is not registered in the asynchronous domain. While the counter increments, the XOR outputs can glitch. If the first synchronizer flip-flop samples during that window, it can capture a value that is off by more than one bit [8]. The results in Chapter 7 indicate that the hazard had a minimal, rather than severe, impact. But still worth noting and investigated as a potential cause for some misattributed counts. A possible solution, as mentioned in the Stanford lecture, is adding a register within the Gray code section; something that can be worked on in the future design.

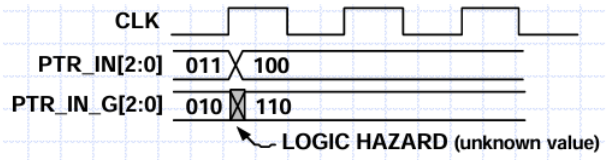


Figure 3.6: Logic hazard from combinational logic [8]

With the help of Vivado's toolset, the AXI-Lite reads out the following registers mapped addresses. The addresses are offset by the RTL's base address of 0x43C0_0000. A comparable AXI-Lite register interface was provided to help navigate the address editing [9].

Address Offset	Register	Description
0x14	ADDR_CTRL	Control
0x18	—	Reserved (unused legacy register)
0x1C	ADDR_COLLISION	SigGen expected collision count
0x20	ADDR_GRANT_CH0	Channel 0 count
0x24	ADDR_GRANT_CH1	Channel 1 count
0x28	ADDR_GRANT_CH2	Channel 2 count
0x2C	ADDR_GRANT_CH3	Channel 3 count
0x30	ADDR_GEN_CTRL	SigGen control
0x34	ADDR_GEN_PERIOD	SigGen period (clock cycles)
0x38	ADDR_GEN_PHASE	SigGen phase pattern
0x3C	ADDR_GEN_NEV	SigGen events per channel

Table 3.1: AXI-Lite register map

3.5 Single-Source Build Strategy

The intention of using a single-source build is to reinforce fairness during comparison. The idea is that every build shares an identical source, everything from the signal generator to the AXI-Lite interface. This minimizes differences when comparing the various designs being tested. However, identical sources does not produce identical placement. Each build is placed and routed separately after implementation is run. So the shared logic lands in different locations with different routing delay. As we progress, Chapter 7 Section 7.1 show these placement differences and measurements.

3.6 Design Matrix

Design	Arbitration	Channels	Async Counters	Notes
No Arbiter	OR merge	4	1	Baseline
FPA Cell / RRA Cell	Ch0 > Ch1; Ch2 ↔ Ch3	2 + 2	2	Two cells, one build
FPA Tree	(Ch0 > Ch1) > (Ch2 > Ch3)	4	1	—
RRA Tree	(Ch0 ↔ Ch1) ↔ (Ch2 ↔ Ch3)	4	1	—

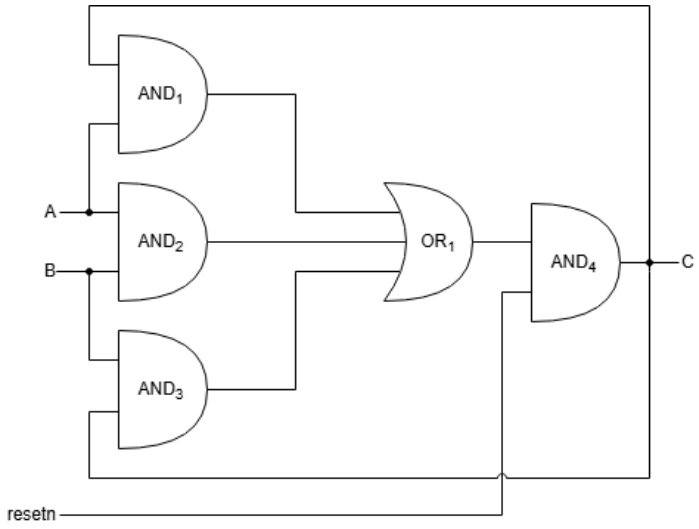
Table 3.2: Design Matrix

When choosing a design from the verilog header file, the choices are between the four seen in Table 3.2. As explained earlier, every change to a new design is a new build which means new placement. However, the logic in arbitration, number of channels and asynchronous counters do not change. What is shared with all designs in the matrix are the handshakes between `req` and `ack`, as well as `grant` and `origin` and `grant_ack`. The primary difference between the designs are its priority networks. To cover it, the no arbiter design does not use a Muller C-element nor a priority scheme. But that is the baseline, and the point of having it. The other designs, however, only within the priority network. This can be seen in Figure 4.3 and Figure 4.4. The Verilog header file is set up globally across the workspace project to update which project in the design matrix is chosen.

```
`ifndef NoArbiter
 `ifndef FPACell
  `ifndef FPATree
   `ifndef RRATree
    `ifndef BothTrees
     `define RRATree             // change this for design
    `endif
   `endif
  `endif
 `endif
`endif
```

Chapter 4: Arbiter Implementation in FPGA Fabric

4.1 Mapping Muller Gates to Look-Up Tables



(a) Gate-level implementation

A	B	C
0	0	0
0	1	C_{prev}
1	0	C_{prev}
1	1	1

(b) Truth table

Figure 4.1: Muller C-element [13] as implemented within the FPA and RRA cells. Resetn is active LOW.

$$C = \text{resetn}\left(AB + AC_{\text{prev}} + BC_{\text{prev}}\right) \tag{4.1}$$

The purpose of the Muller C-element is to retain the previous value of the output when both inputs differ. This is important during handshakes between cells and the asynchronous counter. When an event arrives and a handshake is ready, the Muller C-element proceeds to push the event signal forward. But when a new event arrives and the handshake is not ready, it holds the previous value. Likewise, when there is no new event waiting and the handshake is ready, it still holds the previous value. Otherwise, when there is no event waiting and the handshake is no longer ready, both inputs are low and the output returns to zero, completing the handshake and readying the cell for the next event.

When compiled, the synthesizer places all the logic in one LUT due to there only being four inputs and the `dont_touch` attribute; this is further discussed in Section 4.8. This helps preserve the mapping of the logic and prevent synthesis from merging it with surrounding logic. This is consequential for not causing hazards within the logic. With a single LUT, it evaluates the entire Muller C-element function all at once [13].

4.2 The No Arbiter Module

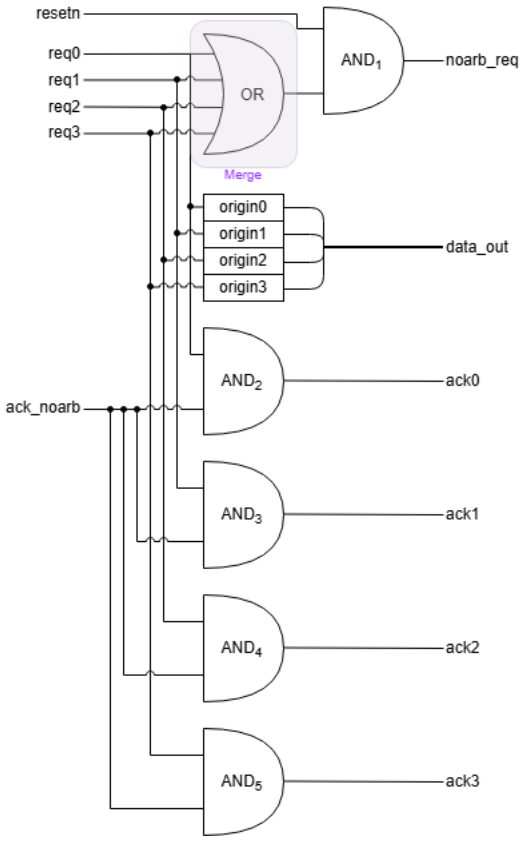


Figure 4.2: No Arbiter Block Diagram

When there are multiple input events arriving at the same time, arbitration becomes critical. Figure 4.2 serves as an example of the absence of arbitration. Its inputs are the all four `req` lines from the asynchronous flip-flops, `ack_noarb` from the asynchronous counter, and `resetn` as the combined hardware and software reset. As any `req` comes in, they will be merged into one output from the OR gate into the AND gate that is already 1'b1 from the

`resetn`; so it's treated almost like a wire. The origin is decoded based on which `req` channel it is. And the `ack_noarb` comes back to handshake it was received by the asynchronous counter. Lastly, it will send an `ack` back to the asynchronous flip-flop based on its the `req` channels origin. The no arbiter design is used as a baseline to show the consequence of not having one. Without arbitration, signals arriving at the same time will merge together, creating one signal. In an environment where almost every photon must be accounted for, this is the least ideal design choice.

4.3 The 2-to-1 Fixed-Priority Arbiter Cell

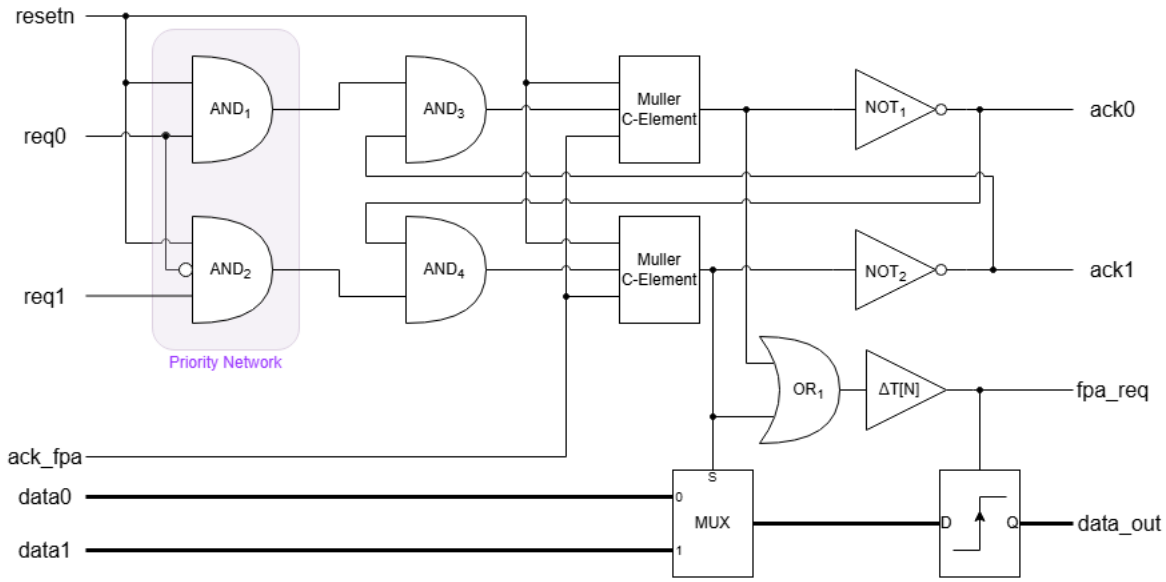


Figure 4.3: FPA Cell (2×1) Block Diagram

As a call back to the design seen in Figure 2.2 [16], a similar design was created to test. Despite the design being similar, Figure 4.3 highlight minor modifications to better align it with FPGA logic. Even so, it retains its core identity as a FPA cell. The input `resetn` is the combined hardware and software reset. The inputs `req0` and `req1` are from the asynchronous flip-flops which contain the channel 0 and channel 1's photon events, respectively. The `ack_fpa` is from the handshake response from the asynchronous counter which handles the second and fourth phase of the four phase handshake between the two

modules. The data0 and data1 hold the req events origin. If the FPA is either a single cell or a leaf cell, then data0 and data1 are constant values addressed to the events origin. Otherwise, if it is a parent FPA cell, then the data_out of the child cell is the data input lines. The outputs ack0 and ack1 are the second and fourth phases of the handshake between the arbiter and the asynchronous flip-flops to clear it for more oncoming events. The fpa_req is the first and third phases on the handshake between the arbiter and asynchronous counter. Lastly, the data_out is the origin channel related to the ongoing signal event.

The easiest way to look at Figure 4.3 is by dividing it into two: the MUX and D flip-flop are strictly for data origin, above that is the req event signals except the two NOT gates (NOT_1 and NOT_2). When a req0 or req1 event arrives, it will first go through the priority network. The boolean algebra for both channel priorities are in Equations 4.2 and 4.3. If you treat resetn as always being 1'b1, we can see that P_0 will always take priority during contention of the two channels.

$$P_0 = \texttt{resetn} \cdot \texttt{req0} \tag{4.2}$$

$$P_1 = \texttt{resetn} \cdot \overline{\texttt{req0}} \cdot \texttt{req1} \tag{4.3}$$

After the priority network selects the winning channel, or req line, it is sent through a cross-coupled structure analogous to a NAND SR latch with the Muller C-element in between for each arm. This allows one req to complete a grant without the other interrupting. As explained in Section 4.1, the value is held as a means for memory until it is cleared by the ack_fpa handshake received from the asynchronous count. The req is sent through the OR_1 gate and two ΔT buffers which serves as a knob to give the data origin enough time to arrive to the D flip-flop. The fpa_req is sent forward either to parent cell or the asynchronous counter as a grant that successfully got arbitrated. And the ack0 and ack1 signals are sent to the asynchronous latch to allow more event signals to arrive.

The last minor change that was made between Figure 2.2 and Figure 4.3 was the select wire to the MUX for `data` origin. With ASIC, designers are able to control the delay. Unfortunately, the delay with FPGAs cost LUTs, which cost space. Instead of using the exact same schematic as seen in Figure 2.2, the wire was moved upward to reduce latency between the two signals required to allow `data_out` to output on time.

4.4 The 2-to-1 Round-Robin Arbiter Cell

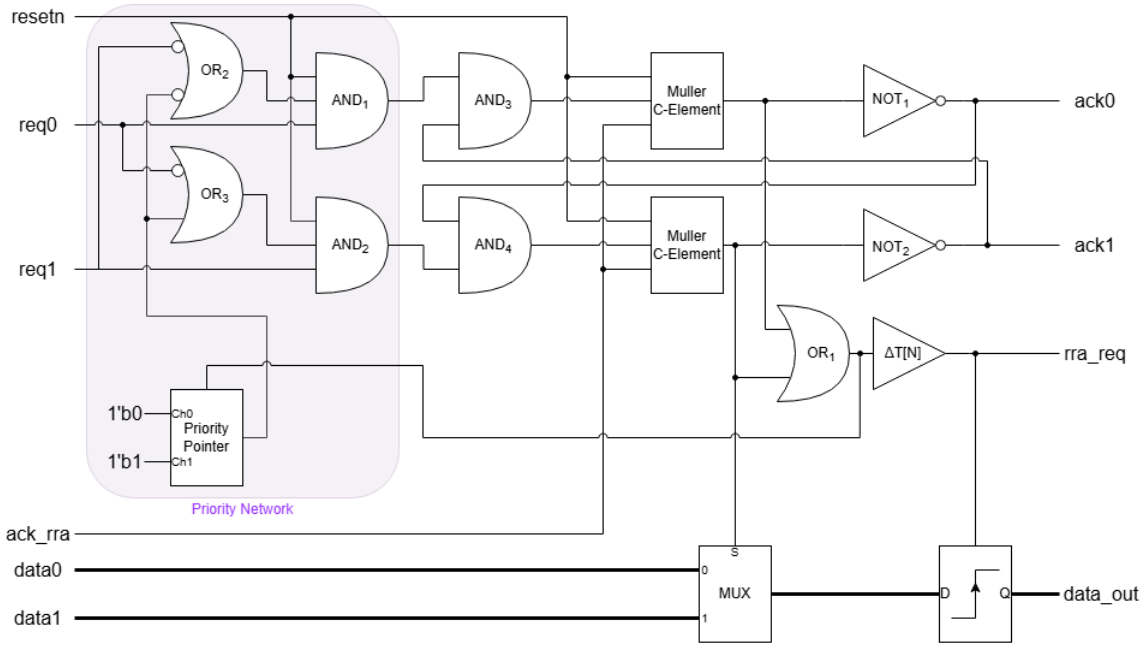


Figure 4.4: RRA Cell (2×1) Block Diagram

Similar to the FPA cell, the RRA cell still arbitrates incoming event signals as needed, as shown in Figure 4.4. Because the design is the exact same outside the priority network, no further explanation on inputs and outputs, as well as other gate logics as they all remain the same as Figures 4.3. The core behavior of the RRA cells priority is to rotate every time an event arrives where the priority pointer is pointing at. Even when there is no contention between channels and an event arrives, the priority pointer will switch. The goal of this priority network is to achieve fairness by using a rotating pointer. The pointers addition to the priority networks equation can be seen in Equations 4.4 and 4.5.

$$P_0 = \texttt{resetn} \cdot \texttt{req0} \cdot \left(\overline{p} + \overline{\texttt{req1}}\right) \tag{4.4}$$

$$P_1 = \texttt{resetn} \cdot \texttt{req1} \cdot \left(p + \overline{\texttt{req0}}\right) \tag{4.5}$$

4.5 Fixed-Priority Arbiter Tree

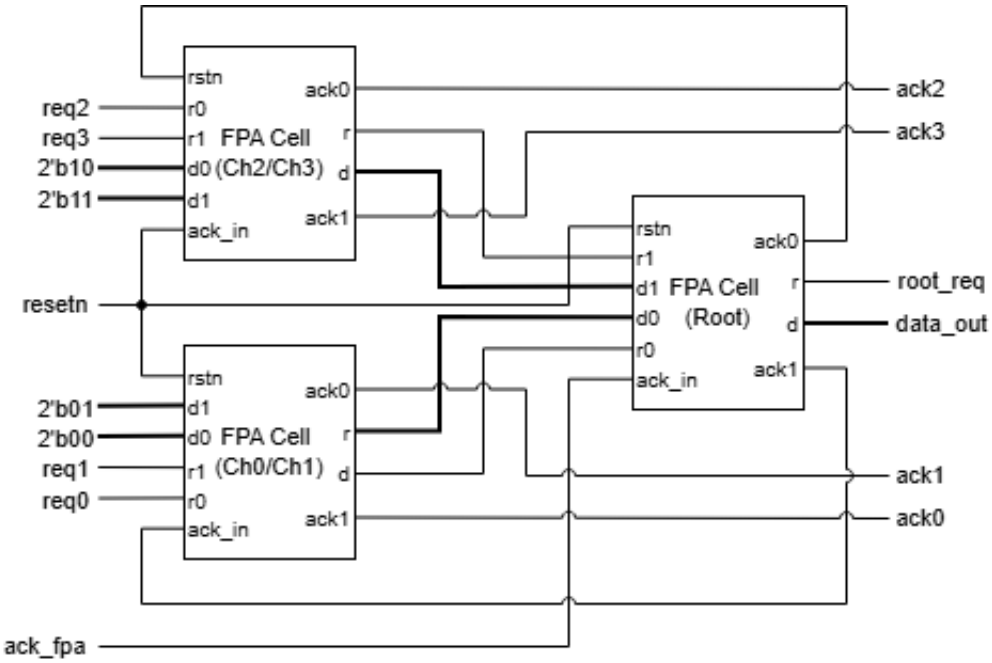


Figure 4.5: FPA Tree (4×1) Block Diagram

The FPA tree uses four channels total, two channels per cell. This is seen in Figure 4.5. Each cell performs the exact logic seen earlier in Figures 4.3. As explained earlier in Section 4.3, the child cells take in the binary values of its corresponding `req` signal, and the parent cell (or root) takes in both the child cells `data_out` as its `data0` and `data1` inputs. This is how the FPA tree's 2-bit origin gets build. Compared to the FPA cell design, the FPA tree handles the handshaking between the arbiter and asynchronous counter at the root cell, while the child cells handle the handshaking between the arbiter and the asynchronous flip-flop. To complete the chain of handshaking, a link from the parents `ack` output to the

child cells `ack_in` (`ack_fpa` or `ack_rra`). This additional handshake is required between cells when adding more N levels to the tree. Another additional change when creating the FPA tree was halving the amount of nested buffer delays in ΔT at the root cell. During experimentation, it was observed that by removing one buffer from the root shortened the handshake enough that some lost events under a four-way contention no longer occurred.

4.6 Round-Robin Arbiter Tree

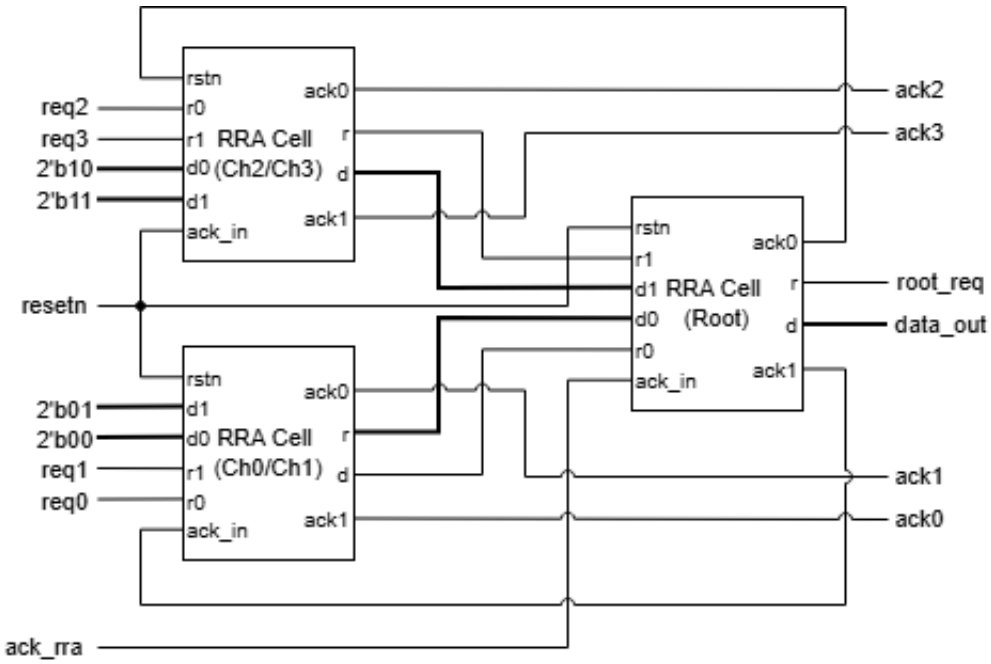


Figure 4.6: RRA Tree (4×1) Block Diagram

The FPA tree's structure is similarly setup like the RRA trees. Looking at Figure 4.4, all three cells in Figure 4.6 are the same internally. As mentioned previously when discussing the FPA tree, the RRA tree's root buffer ΔT is halved. The same positive affect also affected the RRA tree's total event count. The experiments were done on the bench because it was difficult to simulate hardware buffers in logic, which is why they were called knobs as it is a means to tune for timing.

4.7 Delay Chains as Tuning Parameters

As shown in Figure 3.4 and Figure 4.3 or 4.4, there are four LUT buffer chains. Each buffer assists in controlling timing between handshakes. The origin buffer assists with the origin flip-flop to ensure the origin data registers with enough time. The ack buffer assists with timing the `grant_ack` signal during the handshake back to the arbiters. The count buffer assists with the count register to ensure the origin data gets there with enough time. The ΔT assists with the D flip-flop at `data_out`.

4.8 Constraining Intentional Combinational Loops

When using Vivado, hardware loops are by default forbidden by the DRC when attempting to generate a bitstream. It also trim or restructure loop logic because it interprets the clockless feedback paths as unstable logical contradictions. The designs seen in this project all have hardware loops, either in the muller c-element to hold the state, the handshake between the arbiter to the asynchronous counter and back, or the nested handshake between the parent and child cells. This is intentional as it is required for asynchronous designs to work. To bypass their strict design rules, explicitly tell the software to allow combinational loops as seen below.

```
# allows looping in the arbiters
set_property ALLOW_COMBINATORIAL_LOOPS TRUE [get_nets -of_objects \
    [get_cells -hier -filter {NAME =~ *arbiters*inferred*}]]
```

Placing this inside an XDC within the project workspace, specifically targeting the cells affected, will cause Vivado to ignore it. Another technique is to apply `dont_touch` attribute in the RTL itself. This prevents both synthesis and implementation from optimizing away or flattening the arbiter logic [18]. The attribute matters because without it the cross-coupled gates seen in the Muller C-elements may merge or remove any of the LUTs used for it, causing the arbiter to not behave as designed. However, it should be let known that this only stops Vivado from optimizing across the preserved logic, not where the logic is placed

or how it is routed. An example of DONT_TOUCH seen in the FPA Cell Verilog code below.

```
(* dont_touch = "true" *) wire r0_in, r1_in;
(* dont_touch = "true" *) wire a0, a1;
```

4.9 Timing Closure Across the Design Matrix

	Slack (ns)	
Build	WNS	WHS
No Arbiter	3.300	0.037
FPA + RRA Cells	3.423	0.037
FPA Tree	3.339	0.037
RRA Tree	3.212	0.037

Table 4.1: Post-implementation timing closure per build.

Vivado generates a post-implementation timing summary report every hardware rebuild using the Static Timing Analysis (STA). Because the STA evaluates the timing analysis on the synchronous portion of the code, the asynchronous logic is excluded with these results as it has no clock to tie itself to. So, the delay within the arbiters are extracted from post-route path reports shown in Table 7.2 in Chapter 7. But the timing closure here is still important to mention as it does indicate that the synchronous portions of the code, which are the synchronous counters, AXI-Lite, and signal generator, all met timing. The signal generator creates the in-fabric events that is tested later. The synchronous logic gets carried over to software. So meeting timing means both of these can be trusted with every build shown. Table 4.1 shows the Worst Negative Slack (WNS) and Worst Hold Slack (WHS) for the four tested builds.

Chapter 5: Behavioral Verification

A behavioral simulation is used to verify the arbiter designs' logic. The four designs tested are: FPA Cell (2×1), RRA Cell (2×1), FPA Tree (4×1), and RRA Tree (4×1). The goal of every test is to inspect the arbiter's behavior in every condition. The testbench drives the arbiter through the baseline, arbitration, timing, and reset conditions, as seen in 5.1. In every testbench, the channel latches and asynchronous counters are modeled to respond to each design's outputs.

```
1 // Asynchronous channel flip-flops handshake
2 always @(negedge ack0) req0 <= #0.5 1'b0;
3 always @(negedge ack1) req1 <= #0.5 1'b0;
4 // Asynchronous counter handshake
5 assign #1 ack_fpa = ~fpa_req;
6 // Counters
7 always @(negedge ack0) cnt0 = cnt0 + 1;
8 always @(negedge ack1) cnt1 = cnt1 + 1;
```

The choice to show behavioral simulation over timing is due to buffer approximation at ΔT, and hardware loops. Logic buffers use LUT1 which can not be accurately simulated due to placement delay. Since each build placement may vary, the decision was to focus more on the actual hardware tests. Also, combinatorial and asynchronous hardware loops make for unpredictable simulation and synthesis mismatch. So, a behavioral simulation is shown in this chapter to verify logic and protocol in both FPA and RRA designs. It may be worth noting that a behavioral timing analysis was done on both arbiters and passed all tests.

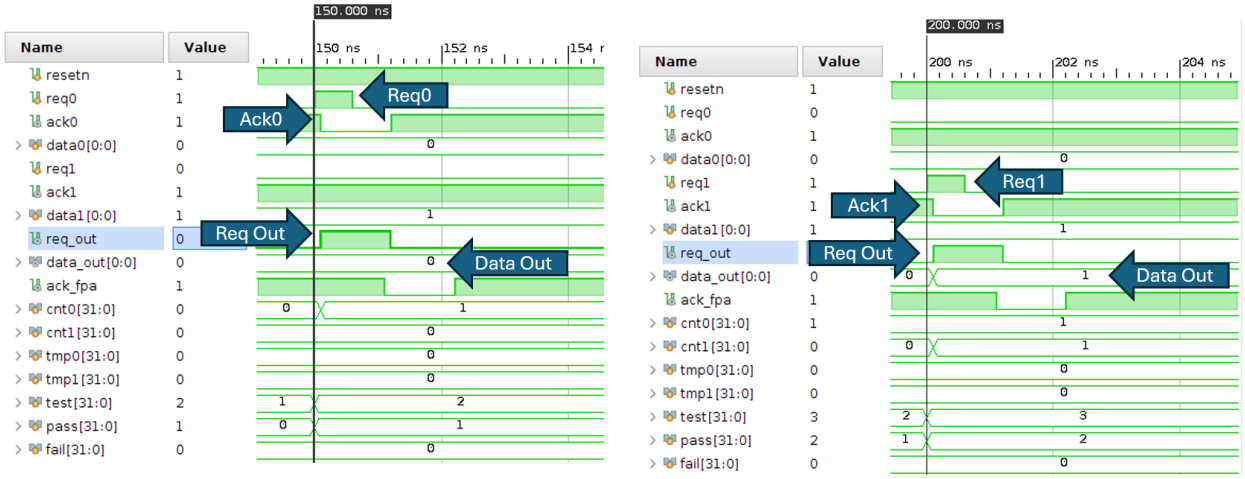
5.1 Unit-Cell Testbench

Test	Test Vector	Expected Result	Result	
			FPA	RRA
Baseline				
1	Hold `rstn` low for 100 ns, then release	Arbiter idles; `req_out` = 0	✓	✓
2	Single `req0` pulse	One completed cycle; `data_out` = `data0`	✓	✓
3	Single `req1` pulse	One completed cycle; `data_out` = `data1`	✓	✓
4	Two `req0` pulses, 50 ns apart	Ch0 counted twice	✓	✓
5	Two `req1` pulses, 50 ns apart	Ch1 counted twice	✓	✓
Arbitration				
6	`req0` and `req1` simultaneously	Both counted once; Ch0 first, Ch1 last	✓	✓
7	`req1`, then `req0` 0.5 ns later	Both counted once; Ch1 first, Ch0 last	✓	✓
8	`req0`, then `req1` 0.5 ns later	Both counted once; Ch0 first, Ch1 last	✓	✓
Timing				
9	`req0` re-asserted 0.5 ns later (still high)	Merged; Ch0 counted once	✓	✓
10	`req1` re-asserted 0.5 ns later (still high)	Merged; Ch1 counted once	✓	✓
11	Two `req0` pulses 42 ns apart (COUNT-100C dead time)	Ch0 counted twice	✓	✓
12	Two `req1` pulses 42 ns apart (COUNT-100C dead time)	Ch1 counted twice	✓	✓
Reset				
13	`rstn` asserted 0.5 ns into a contended transaction	`req_out` = 0 within 5 ns	✓	✓
14	`rstn` released after Test 13	Remains idle	✓	✓
Priority Specific				
15	FPA: `req0` pulse then `req0` and `req1` pulses simultaneously	Ch0 keeps priority; Ch0 first, Ch1 next	✓	—
16	RRA: `req0` pulse then `req0` and `req1` pulses simultaneously	Ch0 gives up priority; Ch1 first, Ch0 next	—	✓

Table 5.1: Behavioral verification of the FPA Cell (2×1) and RRA Cell (2×1). The ✓ means the test passed in behavioral simulation, and the — means it is not applicable.

Waveforms of tests 2, 3, 6, 15, and 16 in Table 5.1 are shown in Figures 5.1 - 5.4. The takeaway from the baseline is that each arbiter cell's reset worked properly, and both channels were able to count under no contention. From the arbitration, both arbiters were able to resolve a simultaneous request as well as a near same-time arrival by 0.5 ns. Next

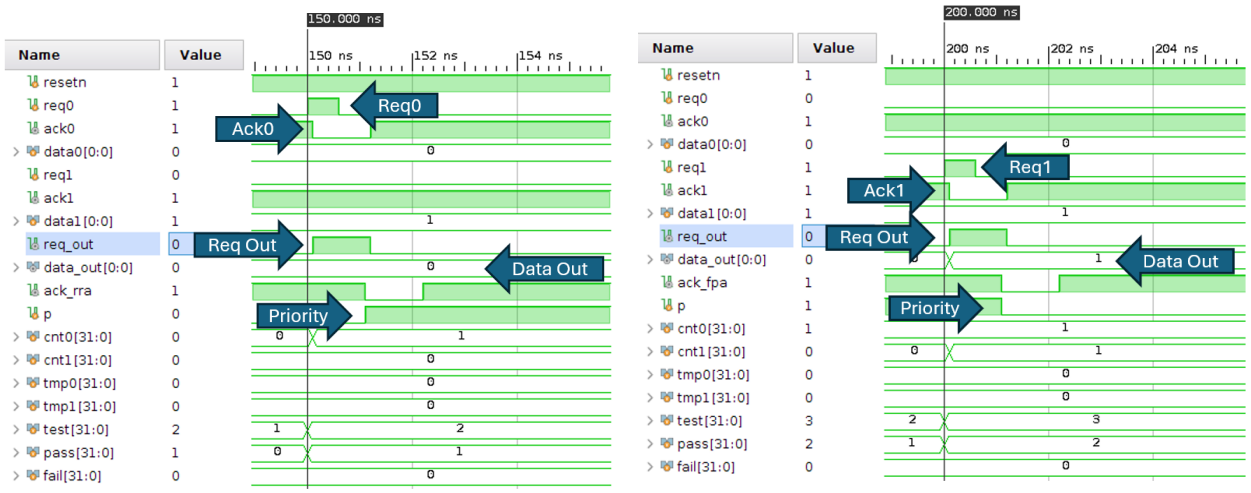
is timing, the goal of tests 9 and 10 was to view the behavior of of the arbiter if either `req0` or `req1` were to receive an event arrival too fast; which shouldn't happen due to the handshaking policy between the channel flip-flops. As expected, both ch0 and ch1 counted only once despite receiving two requests. Tests 11 and 12 using COUNT-100C's pulse and dead time was also used to verify logic behavior. Tests 13 and 14 to cover more with reset to ensure there were no abnormalities in logic values switching incorrectly. And finally, a tests 15 and 16 checked priority was working properly for both cells.



(a) Test 2: single `req0` signal **(b)** Test 3: single `req1` signal

Figure 5.1: FPA Cell (2×1) behavioral simulation of tests 2 and 3.

Figure 5.1 examines the waveform results of tests 2 and 3 on the FPA cell. The primary logic to look out for are the `req` and `ack` input signals, and `req_out` and `data_out` output signals, as annotated. Both `req0` and `req1` are inputs from the testbench, but are switched low 0.5 ns after `ack` receives it. This is proper behavior as it is described in the testbench code above. In Figure 5.1b, `data_out` can be seen switching from 1'b0 to 1'b1 to reflect the incoming signals channel of origin. These two tests verify each channels logic for the FPA cell.

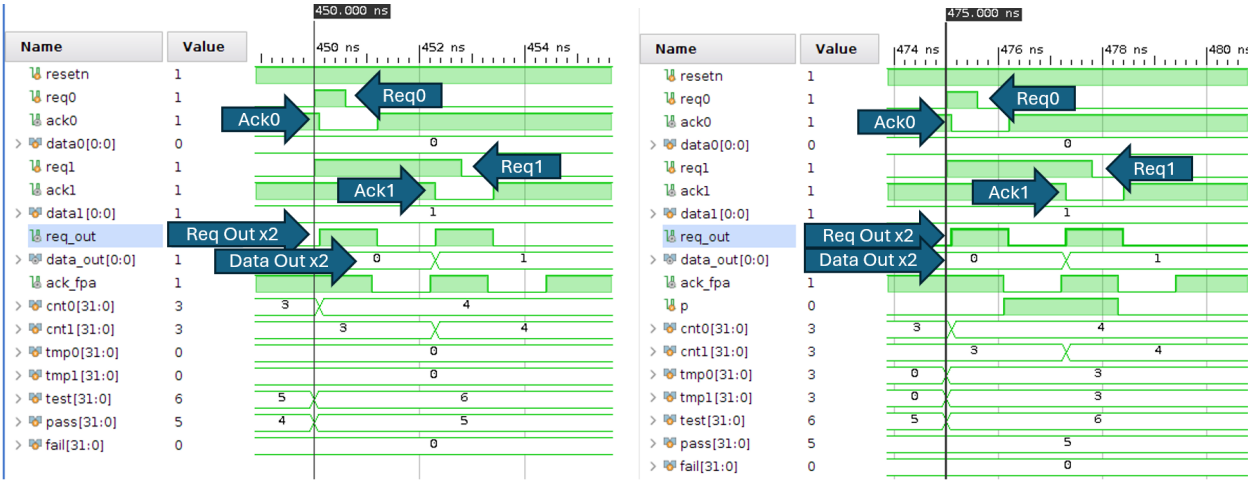


(a) Test 2: single `req0` signal

(b) Test 3: single `req1` signal

Figure 5.2: RRA Cell (2×1) behavioral simulation of Tests 2 and 3.

Similarly to the FPA cell, the RRA cell's test results show the same behavior for tests 2 and 3, but with the inclusion of the internal rotating priority signal `p`. As shown in Figure 5.2, both inputs `req0` and `req1` were able to go through the RRA cell and get cleared by the pseudo channel flip-flop. Both `req_out` and `data_out` outputs also behaved similarly to the FPA cells result and tagging the correct channel origin. The `p` switched from low-to-high in Figure 5.2a or high-to-low in Figure 5.2b after the request was received on the channel with priority.

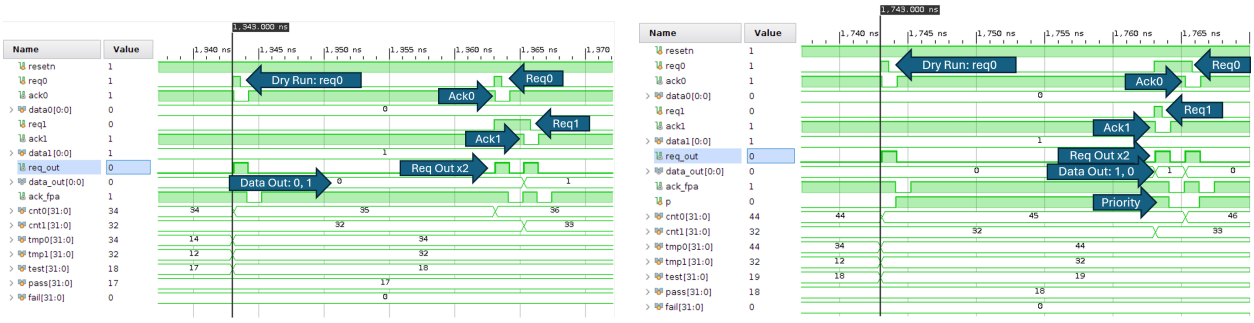


(a) Test 6: FPA cell

(b) Test 6: RRA cell

Figure 5.3: Both FPA and RRA Cells (2×1) behavioral simulation of Test 6. Both arbiters resolve the simultaneous arrival successfully.

Knowing that single-channel throughput logic works, testing simultaneous arrival is next, and the primary job of the arbiters itself. Taking a look at Figure 5.3, both FPA and RRA cells are placed next to each other for comparison. Both cells took the two requests simultaneously and handled each of them the same. First `req0` is sent through the arbiter, followed by `req1`. The output `req_out` can be seen sending the two signal events out sequentially with `data_out` tagging the correct origin per event. The primary difference is when priority `p` switches twice. Before this test started, the RRA cell was reset, which means `p` prioritizes `req0` first by default. Because the two requests arrived at the same time it prioritized `req0` first, rotated priority, then prioritized `req1`, then rotated priority again. As an example, if only `req0` arrives while `p` is pointing at `req1`, `req0` still goes through the arbiter but the priority does not rotate. However, if only `req1` arrives while the `p` is pointing at `req1`, it will still rotate priority, as shown in Figure 5.2b.



(a) Test 15: FPA Cell **(b)** Test 16: RRA Cell

Figure 5.4: Both FPA and RRA Cells (2×1) behavioral simulation of Tests 15 and 16.

To ensure priority for both still work, a dry run of `req0` was sent to both arbiter cells. Afterwards, a simultaneous request arrival was sent. The expected result for the FPA cell is for it to send the requests out in the order of `req0` and then `req1`. The FPA cell succeeded. The expected result for the RRA cell is for it to send the requests out in the order of `req1` and then `req0`. The reason is because priority rotated from `req0` to `req1` after the dry run. The RRA cell succeeded. This concludes the verification of the logic behavior in both the FPA and RRA cells.

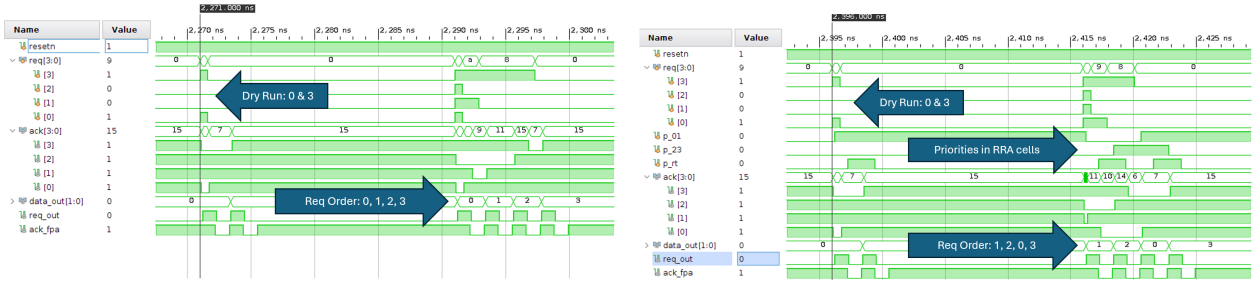
5.2 Tree-Level Testbench

Test	Test Vector	Expected Result	Result	
			FPA	RRA
Baseline				
1	Hold `rstn` low for 100 ns, then release	Tree idles; `req_out` = 0	✓	✓
2	Single `req[0]` pulse	Ch0 counted once; `data_out` = `00`	✓	✓
3	Single `req[1]` pulse	Ch1 counted once; `data_out` = `01`	✓	✓
4	Single `req[2]` pulse	Ch2 counted once; `data_out` = `10`	✓	✓
5	Single `req[3]` pulse	Ch3 counted once; `data_out` = `11`	✓	✓
Arbitration				
6	All four channels simultaneously	Each counted once; order 0, 2, 1, 3	✓	✓
7	Ch1 and Ch2 simultaneously (across leaf cells)	Each counted once; order 1, 2	✓	✓
8	Ch0 and Ch3 simultaneously (across leaf cells)	Each counted once; order 0, 3	✓	✓
9	Ch0 and Ch1 simultaneously (same leaf cell)	Each counted once; order 0, 1	✓	✓
10	Ch2 and Ch3 simultaneously (same leaf cell)	Each counted once; order 2, 3	✓	✓
Timing				
11	All four pulsed every 42 ns for 20 cycles (COUNT-100C dead time)	Each channel counted 20 times; no events lost	✓	✓
Reset				
12	`rstn` asserted 1 ns into four-way contention	`req_out` = 0 within 5 ns	✓	✓
13	`rstn` released after Test 12	Remains idle	✓	✓
Priority Specific				
14	FPA: `req0`/`req3` pulse then `req0`/`req1`/`req2`/`req3` pulses simultaneously	Order: 0, 1, 2, 3	✓	—
15	RRA: `req0`/`req3` pulse then `req0`/`req1`/`req2`/`req3` pulses simultaneously	Order: 1, 2, 0, 3	—	✓

Table 5.2: Behavioral verification of the FPA Tree (4×1) and RRA Tree (4×1). The ✓ indicates the test passed in behavioral simulation, and the — indicates it is not applicable.

Because the the FPA and RRA cells conclude the core logic behavior, the tree tests the throughput and data arbitration through two layers of cells. So, the takeaway of Table 5.2 is that it can perform single-channel throughputs, resolve simultaneous arrivals, complete arbitration using the COUNT-100C's max throughput, ensure there's no reset abnormalities,

and finally test priority of both arbiter trees. To go over it in-depth, the baseline tests resetn to scan for abnormal behavior at the start. And then single-channel throughput of all four channels to check all packets were able to go through the tree successfully. For the arbitration tests, all four channels were tested simultaneously and all arrived in the expected order: first ch0, then ch2, then ch1, and lastly ch3. The root was then tested in tests 7 and 8, while the child leaf cells were tested in tests 9 and 10. All cell specific tests in the arbiter tree structure passed, and the root outputted the results in the correct order. Timing tests, similarly to the cells tests in Table 5.1, checked to see if the events received at the period of COUNT-100C would still work, and the arbiter trees succeeded. Another reset test for abnormality, similarly in the cells test, no signals showed signs of abnormal behavior during and after reset amid full contention. Finally, the priority specific tests showed expected results in its behavioral logic. Both the FPA and RRA tree's channel origin order was presented correctly. This can be seen in Figure 5.5.



(a) Test 14: FPA Tree **(b)** Test 15: RRA Tree

Figure 5.5: Both FPA and RRA Trees (4×1) behavioral simulation of Test 14 and 15.

When looking at Figure 5.5, observe how the priority in the RRA cells rotate. The priority in leaf cells rotate only twice, while the root rotates four times. This is because each RRA cell holds its own `p` register that holds its priority pointers value based on the previous request it received. This is why the output of the RRA tree differs from the FPA tree, which is why a dry run was conducted first to showcase this. Otherwise, the FPA and RRA tree will produce the same results as shown in test 6, due to the RRA cells default priority being set to `req0`.

5.3 Limits of Behavioral Simulation for Placement-Dependent Logic

Behavioral simulation, as shown in Sections 5.1 and 5.2, proved the arbiter cells and trees work as intended. However, simulating asynchronous logic proves to be difficult without a clock in the design, and the inclusion of hardware loops. When using event edge-triggers for handshaking, the primary bottleneck then becomes the combined total of logic and routing delay. So when a four-phase handshake is being commenced, the primary obstacle is arrival time between each handshake. Off by a little, the whole signal can be lost. This is why tuning buffers were created. A change of value in N within the chain of buffers will either increase or decrease the sequence of buffers, as discussed in Section 4.7. The thesis compensates for the lack of verification simulations by testing 240 edgecases, max channel throughput, and channel overlapping offset.

Chapter 6: Experimental Methodology

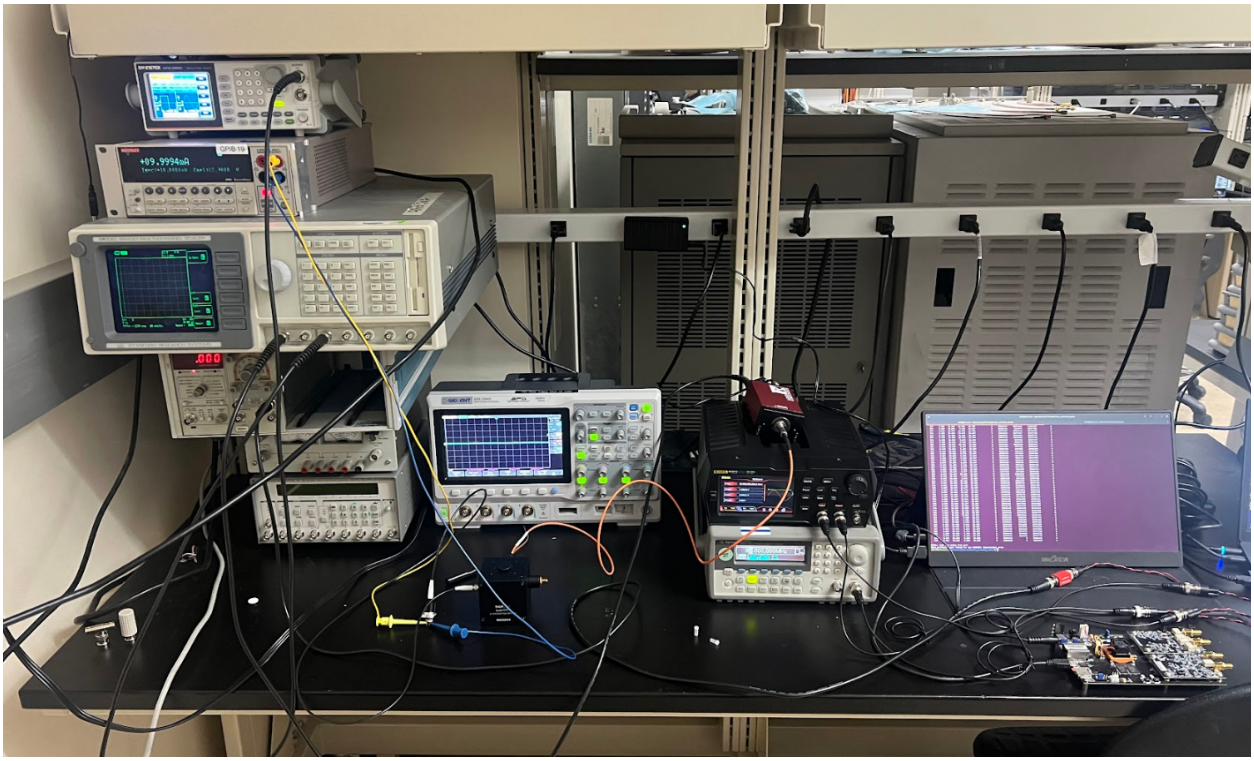


Figure 6.1: Full workstation setup.

Test experiments shown in Sections 6.2 and 6.3 were done in the Science and Engineering Building (SEB), in a lab setting with test equipment and a laptop. Whereas Section 6.1 was done at home, using only a serial port to a computer to view the in-fabric signal generators results. The test equipment used in Section 6.2 are: Stanford Research Systems DG535, Rigol DG992, and an Agilent 33250A pulse generators. The test equipment for Section 6.3 are: Keithley 2400 source-measure unit, Thorlabs pigtailed LED (EP470S04), Thorlabs 2″ integrating sphere (IS236A), Laser Components COUNT-100C SPAD, and Stanford Research photon counters (SR400 and SR430). Both in-lab tests used the Siglent SDS 2304X oscilloscope. All tests are done to prepare the arbiters for all edge cases.

It is important to note that the measured performance could be limited by the para-

sitics of the test set-up. All signals were coupled into the FPGA evaluation board through standard 0.1 inch headers and connected to test equipment with BNC cables. The transition from the BNC cables to the 0.1 inch headers could affect signal integrity and also lead to cross-talk. All bare wires were twisted to attempt to minimize this interference. A high-speed mezzanine port is available on the FPGA but time constraints prevented the development of a custom PCB to plug into this port.

6.1 In-Fabric Signal Generator

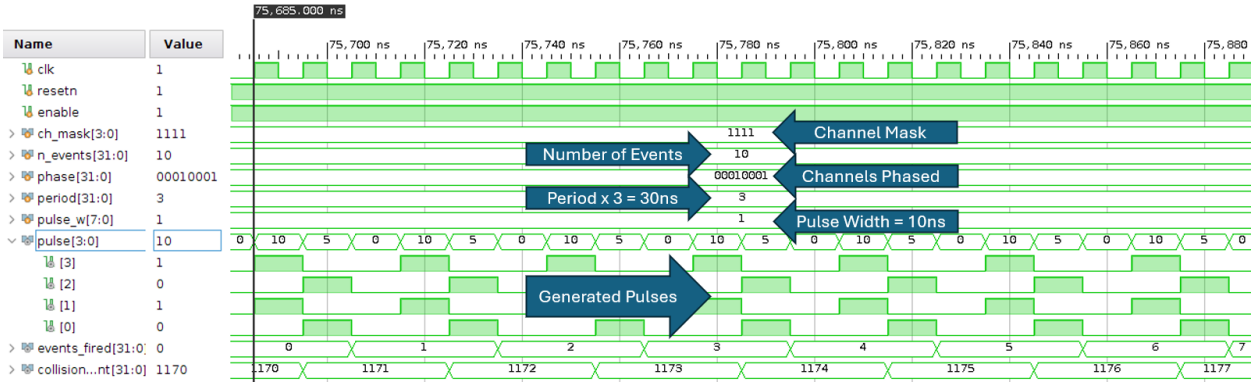


Figure 6.2: Signal generator behavior testbench results.

The in-fabric signal generator has limited ability to create overlapping offset contentions due to the restrictions of outputs being tied to the clock frequency. Since the clock frequency is 100 MHz every unit interval is limited to 10 ns and signals can only have a minimum relative offset of 10 ns. Instead, Section 6.2 has a test using the DG535 that can generate delays with ps level precision. Since the COUNT100C's total period is 42 ns long (10 ns pulse + 32 ns dead time) , it was decided that this test methodology will use a 10 ns pulse width with a variation of periods to see how the arbiter designs respond to full contention, partial contention, and no contention in all possible edge cases. That is 4×4 different active channel configurations (subtract the zero active channels configuration), and 4×4 different phase configurations. That is a total of 240 edge cases.

The period must be greater than or equal to the phase plus the pulse width. Because

the signal generator does not have guard rails to protect against incorrect inputs, it is just manually avoided since the module is used for test purposes only. Otherwise, the pulses can be seen being generated in Figure 6.2, with the five user set configurations. The total count of events fired from the signal generator is also sent upwards to the PS as a truth source to be compared to how many grants were actually counted from the asynchronous counter within the system design.

$$T_{\text{period}} \geq T_{\text{phase}} + T_{\text{pulse_w}} \tag{6.1}$$

6.2 External Pulse Generators

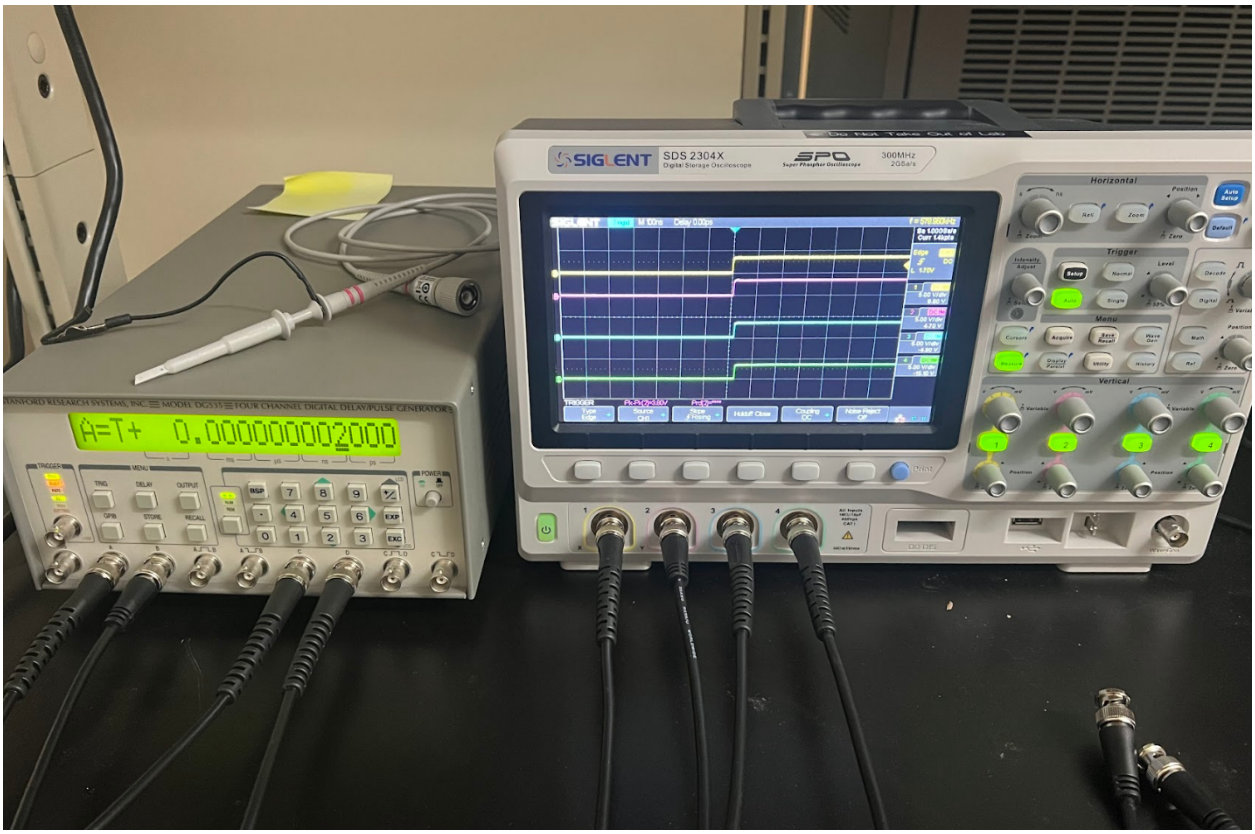


Figure 6.3: Offset testing using Stanford Research Systems delay pulse generator.

To check how the arbiters are affected by slight offsets, Figure 6.3 shows the Stanford Research Systems DG535 (left) setting a delay of 2 ns on port A. The delays chosen to test

with are seen in Table 6.1. For the arbiter cells, only ports A and B will be used, while the trees will use all four ports. The output voltage is 3.3V, and the pulse frequency is 600 kHz. The Siglent oscilloscope (right) shows the slight offset outputted from the DG535.

	Pulse Delay Generator Delay (ns)								
Ports	0	0.25	0.50	0.75	1.00	1.50	2.00	3.00	5.00
A	0	0	0	0	0	0	0	0	0
B	0	0.25	0.50	0.75	1.00	1.50	2.00	3.00	5.00
C	0	0.50	1.00	1.50	2.00	3.00	4.00	6.00	10.00
D	0	0.75	1.50	2.25	3.00	4.50	6.00	9.00	15.00

Table 6.1: DG535 pulse delay generator output delay settings.

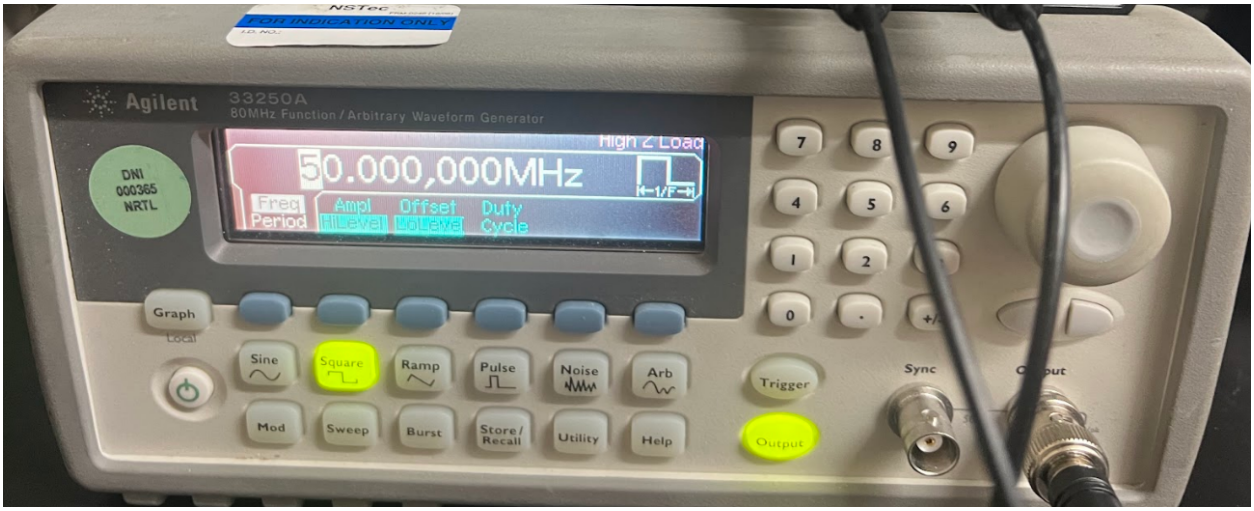


Figure 6.4: Single-channel throughput testing using the Agilent 33250A function generator.

A single-channel throughput speed test was done using the Agilent 33250A function generator because its ceiling frequency is 80 MHz with the 3.3V output required by the FPGA I/O logic. The in-fabric signal generator can only produce pulses up to 50 MHz (a 20 ns minimum period) into the arbiter. At full contention, it was discovered that both the cells and tree's struggled to retain 100% of its channel requests. This test evaluates the throughput efficiency of the arbiters with input event rates well above 50 MHz with no contention. Sixteen target frequencies were used (MHz): 1, 10, 20, 30, 40, 50, 60, 70, 71, 72, 73, 74, 75, 76, 77, and 78. If the in-fabric generator helped establish the frequency limit

at which events are lost during contention, this test will help in discovering the count rate limits when there is no contention.

6.3 SPAD Detector

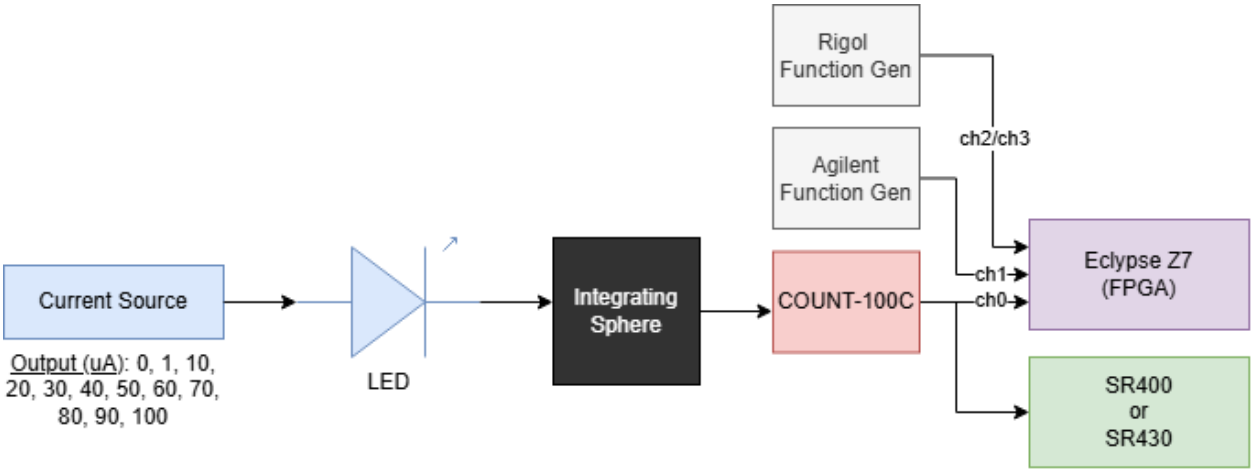
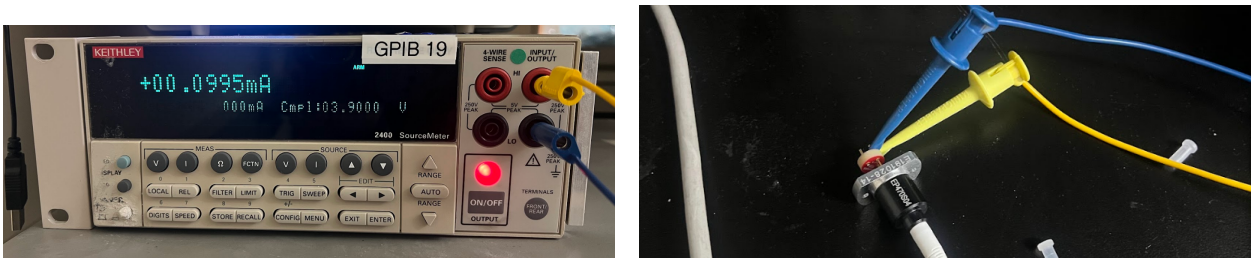


Figure 6.5: Photon counting test setup overview.

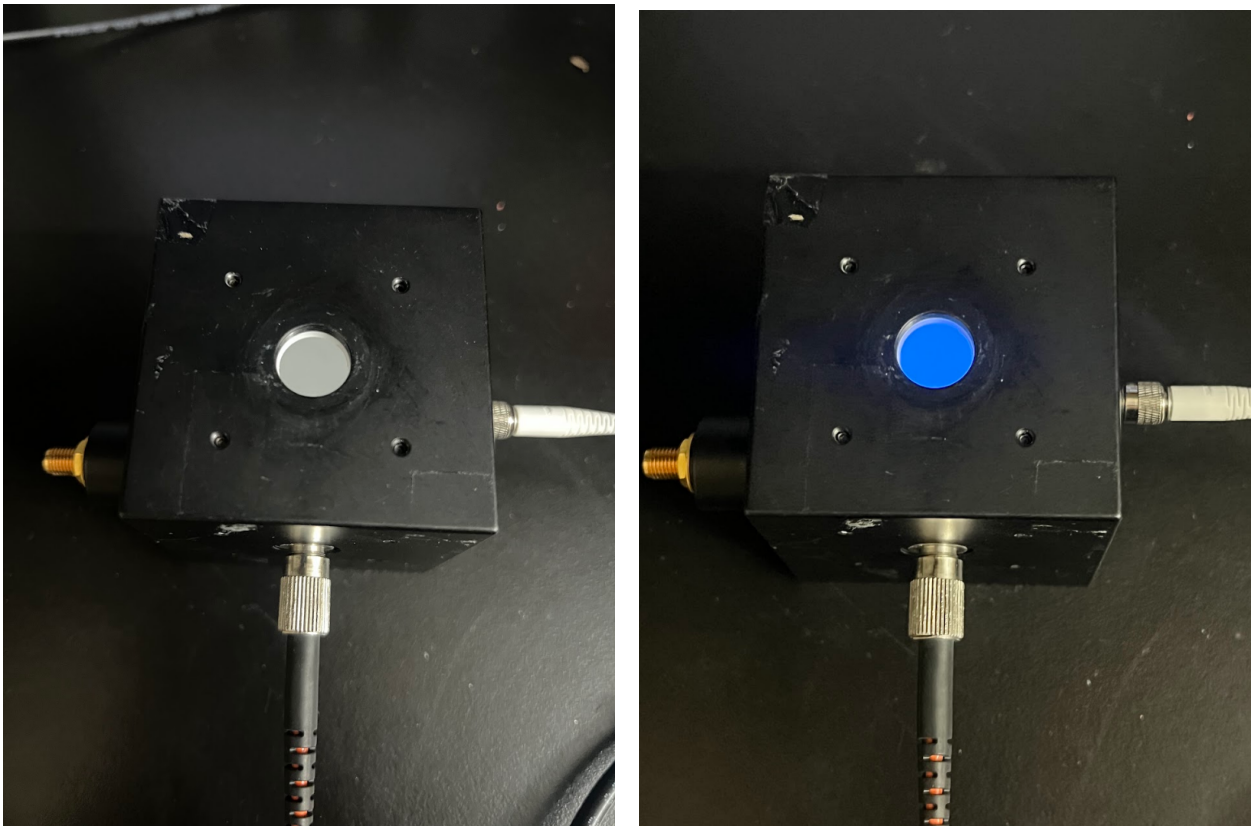
It was not possible to acquire 4 SPAD sources to test the performance of the asynchronous counters with realistic signals. Additionally, 4 reference photon counter instrument channels would be needed to compare results to true photon counts. Instead, 3 out of 4 input signals were from standard function generators and one input signal was from a real SPAD. To introduce a controllable light parameter that the SPAD can input, a current source was used to drive an LED. The currents used during the test can be seen in Figure 6.5. The LED is coupled to a fiber optic patch cable, which is designed to guide the light into the 2″ integrating sphere. From there, a fiber optic cable is attached from the integrating sphere to the COUNT-100C SPAD. Depending on which device is connected to the SPAD, a Transistor-Transistor Logic (TTL) output pulse is sent to either the Eclypse Z7 (arbiters) or the SR400/SR430 photon counters.



(a) Current Source **(b)** LED

Figure 6.6: Current driven LED.

The Keithley power supply can be seen outputting 9.99 mA of current in Figure 6.6a. The LED receives the current source using two banana cable clamps. Yellow as the positive clamped onto the anode (HI), blue as the negative clamped onto the cathode (LO). To prevent the SPAD from saturating, the operating current for this test will not exceed 100 uA. The emitted light is guided from the pigtailed LED assembly into a 2″ integrating sphere via a fiber optic patch cable.



(a) LED with current source of 0 mA. **(b)** LED with current source of 9.99 mA.

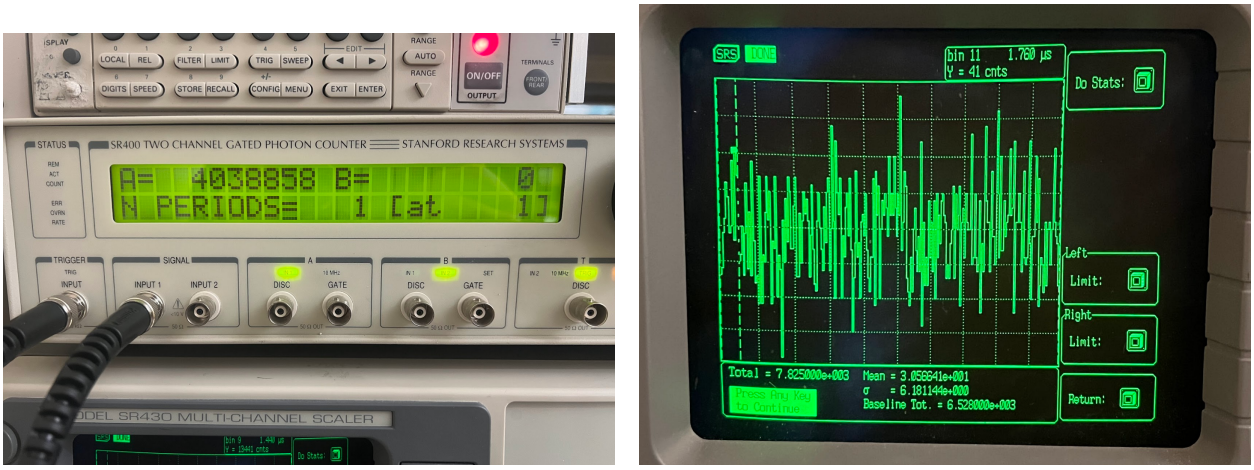
Figure 6.7: Integrating sphere for light optics attenuation.

To verify the internal light optic path in the integrating sphere, a port cap was removed from the top to expose the cavity, as shown in Figure 6.7. The visualization demonstrates the LED's intensity within the integrating sphere under drive currents of 0 mA and 9.99 mA. During actual testing, the sphere is sealed to maintain a light-tight environment within it and preventing light leakage. From the integrating sphere to the SPAD, a fiber optic cable is connected between them. The SPAD will be able to sense the light coming through from the LED.



Figure 6.8: The Count-100C SPAD.

The Laser Components COUNT-100C SPAD has its TTL connected to the integrating sphere, a BNC to a photon counter, and a 12V power outlet. From the SPAD it connects to any photon counter shown in Figure 6.9 and 6.10. The SR430 is set to 400 ms per count, and the SR400 is set to 500 ms per count. Whereas the Eclypse Z7, shown in Figure 6.10, is recording at 1000 ms per count. Both the SR430 and SR400's total count will be multiplied to match the Eclypse Z7's total count for comparison during post test analysis. A similar setup for the SR400 was used in prior work at UNLV [14].



(a) SR400 **(b)** SR430

Figure 6.9: SR400 and SR430 photon counters.

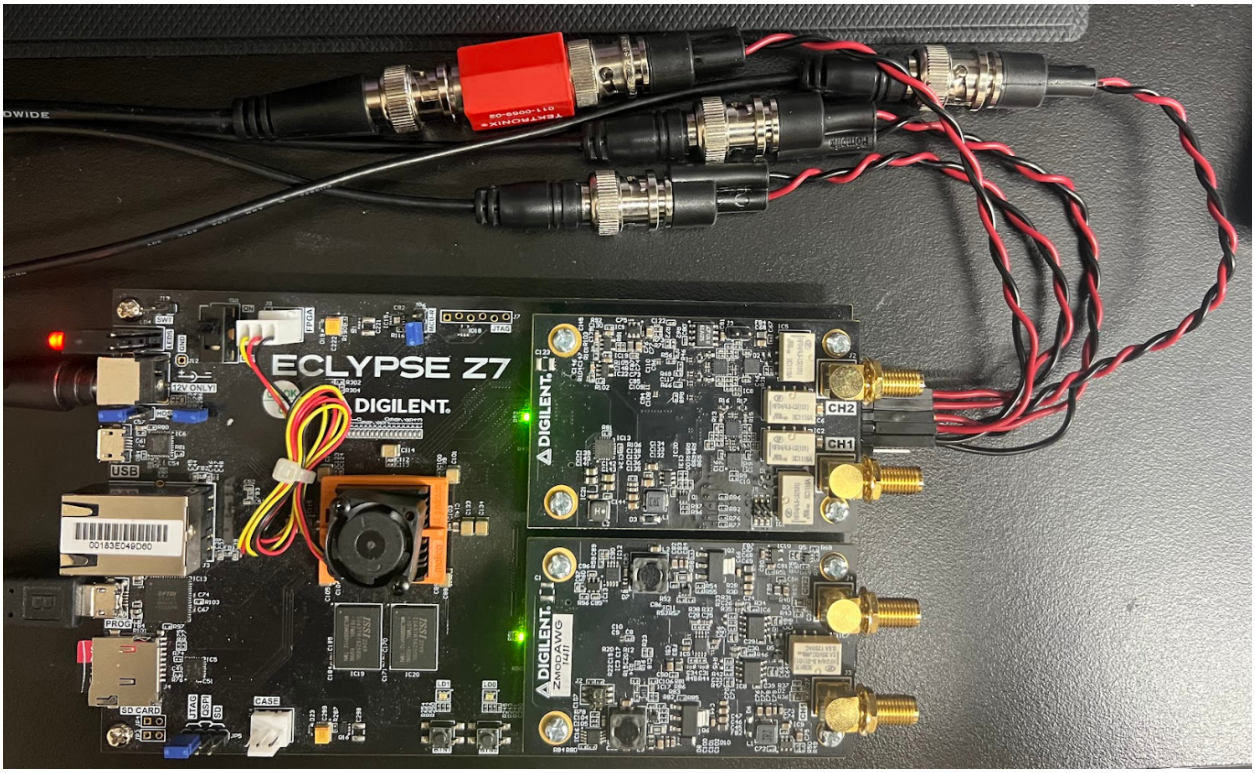


Figure 6.10: Eclypse Z7 wired up for photon testing.

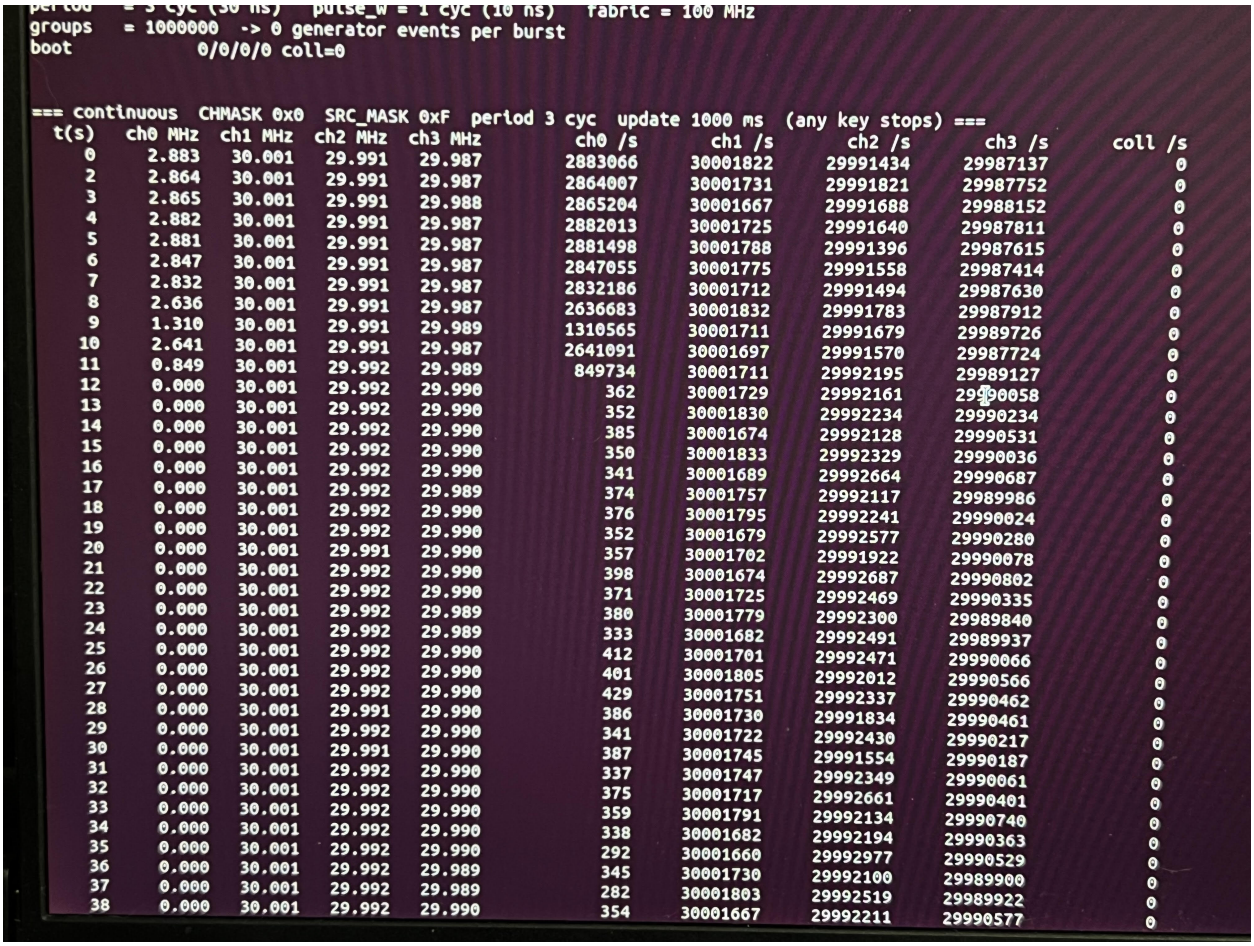


(a) SPAD Readout (Full Light)



(b) SPAD Readout (Partial Light)

Figure 6.11: COUNT-100C SPAD output pulses captured on an oscilloscope



```
groups   = 1000000  -> 0 generator events per burst
boot            0/0/0/0 coll=0

=== continuous  CHMASK 0x0  SRC_MASK 0xF  period 3 cyc  update 1000 ms  (any key stops) ===
  t(s)   ch0 MHz  ch1 MHz  ch2 MHz  ch3 MHz        ch0 /s       ch1 /s       ch2 /s       ch3 /s      coll /s
    0      2.883   30.001   29.991   29.987       2883066     30001822     29991434     29987137            0
    2      2.864   30.001   29.991   29.987       2864007     30001731     29991821     29987752            0
    3      2.865   30.001   29.991   29.988       2865204     30001667     29991688     29988152            0
    4      2.882   30.001   29.991   29.987       2882013     30001725     29991640     29987811            0
    5      2.881   30.001   29.991   29.987       2881498     30001788     29991396     29987615            0
    6      2.847   30.001   29.991   29.987       2847055     30001775     29991558     29987414            0
    7      2.832   30.001   29.991   29.987       2832186     30001712     29991494     29987630            0
    8      2.636   30.001   29.991   29.987       2636683     30001832     29991783     29987912            0
    9      1.310   30.001   29.991   29.989       1310565     30001711     29991679     29989726            0
   10      2.641   30.001   29.991   29.987       2641091     30001697     29991570     29987724            0
   11      0.849   30.001   29.992   29.989        849734     30001711     29992195     29989127            0
   12      0.000   30.001   29.992   29.990           362     30001729     29992161     29990058            0
   13      0.000   30.001   29.992   29.990           352     30001830     29992234     29990234            0
   14      0.000   30.001   29.992   29.990           385     30001674     29992128     29990531            0
   15      0.000   30.001   29.992   29.990           350     30001833     29992329     29990036            0
   16      0.000   30.001   29.992   29.990           341     30001689     29992664     29990687            0
   17      0.000   30.001   29.992   29.989           374     30001757     29992117     29989986            0
   18      0.000   30.001   29.992   29.990           376     30001795     29992241     29990024            0
   19      0.000   30.001   29.992   29.990           352     30001679     29992577     29990280            0
   20      0.000   30.001   29.991   29.990           357     30001702     29991922     29990078            0
   21      0.000   30.001   29.992   29.990           398     30001674     29992687     29990802            0
   22      0.000   30.001   29.992   29.990           371     30001725     29992469     29990335            0
   23      0.000   30.001   29.992   29.989           380     30001779     29992300     29989840            0
   24      0.000   30.001   29.992   29.989           333     30001682     29992491     29989937            0
   25      0.000   30.001   29.992   29.990           412     30001701     29992471     29990066            0
   26      0.000   30.001   29.992   29.990           401     30001805     29992012     29990566            0
   27      0.000   30.001   29.992   29.990           429     30001751     29992337     29990462            0
   28      0.000   30.001   29.991   29.990           386     30001730     29991834     29990461            0
   29      0.000   30.001   29.992   29.990           341     30001722     29992430     29990217            0
   30      0.000   30.001   29.991   29.990           387     30001745     29991554     29990187            0
   31      0.000   30.001   29.992   29.990           337     30001747     29992349     29990061            0
   32      0.000   30.001   29.992   29.990           375     30001717     29992661     29990401            0
   33      0.000   30.001   29.992   29.990           359     30001791     29992134     29990740            0
   34      0.000   30.001   29.992   29.990           338     30001682     29992194     29990363            0
   35      0.000   30.001   29.992   29.990           292     30001660     29992977     29990529            0
   36      0.000   30.001   29.992   29.989           345     30001730     29992100     29989900            0
   37      0.000   30.001   29.992   29.989           282     30001803     29992519     29989922            0
   38      0.000   30.001   29.992   29.990           354     30001667     29992211     29990577            0
```

Figure 6.12: Software Interface (Serial Connected)

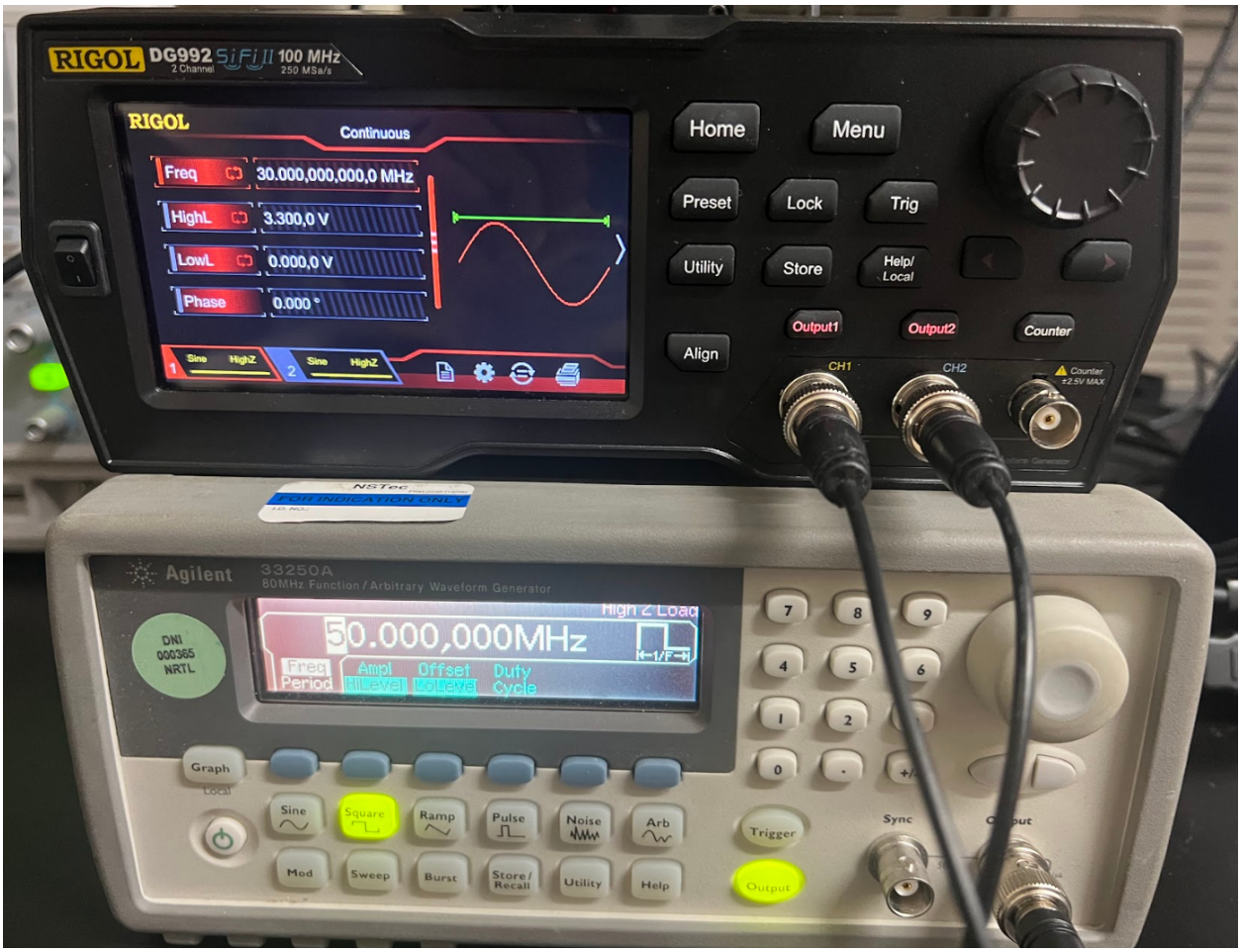


Figure 6.13: Function generators used for 30 MHz contention request signals. Agilent is not depicted in image accurately for this test.

The photon counting for the FPGA will include the SPAD connection alone on ch0, and with contention using the function generators seen in Figure 6.13. The Rigol (top) function generator will be set to a 30 MHz sinusoidal waveform, with low/high levels of 0V/3.3V. The Agilent (bottom) function generator will be set to 30 MHz sinusoidal waveform, with low/high levels of 0V/3.3V as well. The 30 MHz frequency was chosen because it exceeds the SPADs dead-time, so each generating channel is a worst case neighbor providing contention for the SPAD channel.

Chapter 7: Results

7.1 Resource Utilization

The following tables are post-route reports created in Vivado 2019.1, all in a single project, with arbiters selected by a Verilog header file (config.vh), everything else remains identical. This is all explained in Section 3.5. Unfortunately, because the original post-implementation build is lost after every re-build, Table 7.2 does not accurately show exact placement delay. This does not takeaway from the value of rebuilding it and getting a close approximation.

Design	Arbiter LUTs	Custom Logic LUTs	Total Logic LUTs
No Arbiter (4×1)	3	664	1037
FPA Cell (2×1)	9	702	1075
RRA Cell (2×1)	11	702	1075
FPA Tree (4×1)	26	685	1059
RRA Tree (4×1)	31	688	1062
FPA Tree (8×1)	65	928	1303
RRA Tree (8×1)	78	938	1313
FPA Tree (16×1)	125	1418	1793
RRA Tree (16×1)	153	1442	1817

Table 7.1: LUT utilization by design.

The arbiters within the design contain a total of 3-31 LUTs against 1037-1075 in total. This is roughly under 3% of total logic LUT usage across all arbiter variances. So, there is a lot more overhead in the FPGAs infrastructure. The no arbiter build contains barely any LUTs, which is to be expected. More importantly, the RRA cell contains 22.2% more LUTs than the FPA cell, and 19.2% more LUTs at 4×1 (20.0% at 8×1 and 22.4% at 16×1) than the FPA tree. This was also to be expected, as the RRA cell contains two more additional gates and a register compared to the FPA cell. So increasing the levels in the tree will only

further increase the gap between the too as N-levels increase. The custom logic in the FPA and RRA cells are the same because it is in the same build.

Design	Logic Delay (ns)	Routing Delay (ns)	Arbiter Delay (ns)
No Arbiter (4×1)	0.124	1.476	1.600
FPA Cell (2×1)	0.744	3.704	4.448
RRA Cell (2×1)	0.620	2.520	3.140
FPA Tree (4×1)	1.268	7.406	8.676
RRA Tree (4×1)	1.109	6.150	7.259
FPA Tree (8×1)	1.935	7.279	9.214
RRA Tree (8×1)	1.612	7.003	8.615
FPA Tree (16×1)	2.108	9.175	11.283
RRA Tree (16×1)	2.688	11.888	14.576

Table 7.2: Propagation delay through each designs channel 0.

After implementation, a timing report can be generated from one leaf of the design to another. The path from the output of the asynchronous flip-flops to the input of the asynchronous counter is the arbiter delay shown in Table 7.2. This is obtained by taking the total delay it takes to get to the flip-flops output, the total delay it takes to get to the asynchronous counters input, and subtracting the two to get the total delay between. The logic delay is the sum of all the LUT delays attached to the arbiter found in the timing report. Lastly, the routing delay is calculated by taking the arbiter delay and subtracting it to the logic delay.

$$t_{\text{arbiter}} = t_{\text{async_counter}} - t_{\text{async_FF}} \tag{7.1}$$

$$t_{\text{logic}} = \sum_{i=1}^{N} t_{\text{LUT},i} \tag{7.2}$$

$$t_{\text{routing}} = t_{\text{arbiter}} - t_{\text{logic}} \tag{7.3}$$

Build	Power (W)			
	Custom Logic	Dynamic	Static	Total On-Chip
No Arbiter	0.008	1.521	0.140	1.661
FPA + RRA Cells	0.011	1.523	0.140	1.664
FPA Tree (4×1)	0.009	1.521	0.140	1.662
RRA Tree (4×1)	0.009	1.522	0.140	1.662
FPA Tree (8×1)	0.014	1.527	0.140	1.667
RRA Tree (8×1)	0.014	1.526	0.140	1.667
FPA Tree (16×1)	0.017	1.530	0.141	1.670
RRA Tree (16×1)	0.017	1.530	0.141	1.671

Table 7.3: Estimated on-chip power by build.

The power estimation had to be done by build because the FPA and RRA cells are in one build, so there is no way of separating the power consumption usage. Because the FPA and RRA cells are in one build, there are two instantiations of the asynchronous counter, as opposed to one, which shows its impact by having the custom logic at 0.011 W. The designs overall are too small to make a comparison between them. Growing the tree from 4 to 16 channels adds about 8 mW of custom-logic power, roughly 0.5% of total on-chip power. So there is room for growth.

7.2 In-Fabric Signal Generator

The in-fabric signal generator drives all four channels from the FPGA's 100 MHz fabric clock. Each burst emits a known number of pulses per channel, so every count has an exact ground truth. A channel mask selects which channels are active, and a phase pattern selects which active channels are offset by one clock period (10 ns). Together they produce 240 cases: 15 masks × 16 phase patterns. The pulse width is fixed at 10 ns, and the period sweeps between 2, 3, 4, 5, 6, 7, 8, 10, 12, 20, and 40 in 10 ns steps; that is, if the period is 2 that is a 20 ns period. Each configuration was run 3 times. Because the generator is clocked, the arrivals are exactly simultaneous or at least 10 ns apart. As a baseline, each channel

alone conserved every event. The results below show each arbiter's correctness across all channel selections and phase edge cases, as well as its throughput limit across the period sweep.

7.2.1 Two-Channel: No Arbiter vs FPA Cell vs RRA Cell

The channel selections and phase edge case tests how each arbiter resolves requests whether a channel receives an event, or is phase offset, or both. All channels have a pulse width of 10 ns. Figures 7.1 and 7.2 show events lost in percentages on a heat map. The arbiters show that they can handle two events occurring at the same time without the need for an offset. However, having no arbiter show a loss of 50% of events. This is also shown in Table 7.4 in bold. Under identical stimulus, both the FPA and RRA cells recorded no arbitration lost at the same point of failure as the no arbiter design.

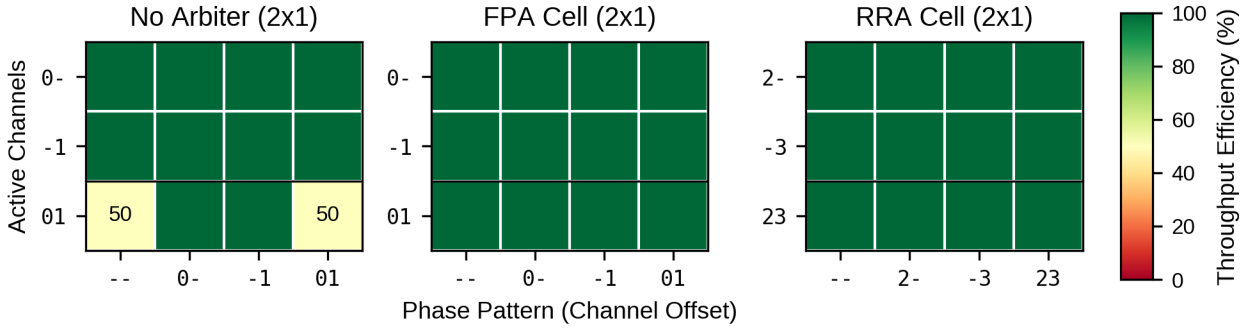


Figure 7.1: Two-channel throughput efficiency by active-channel mask and phase pattern for the No Arbiter, FPA Cell, and RRA Cell designs (period = **30 ns**, 1 million events per channel)

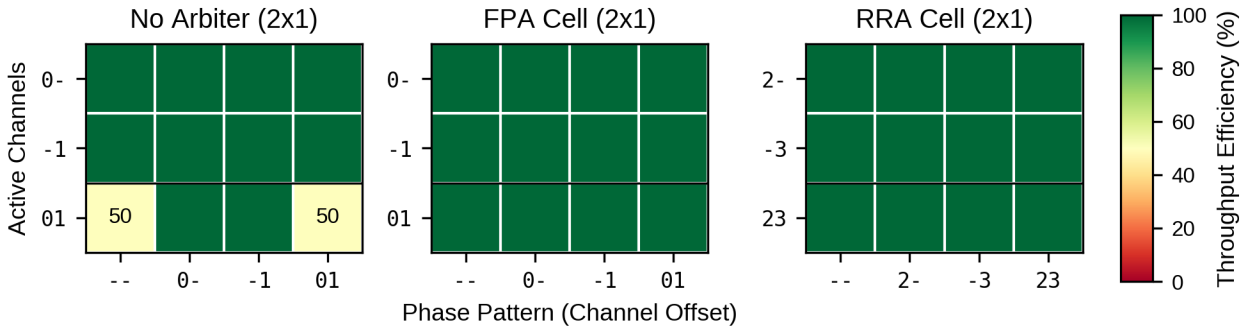


Figure 7.2: Two-channel throughput efficiency by active-channel mask and phase pattern for the No Arbiter, FPA Cell, and RRA Cell designs (period = **20 ns**, 1 million events per channel)

	Period = 30 ns				Period = 20 ns			
Active	None	A	B	Both	None	A	B	Both
No Arbiter								
0-	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
-1	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
01	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**
FPA Cell								
0-	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
-1	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
01	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
RRA Cell								
2-	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
-3	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
23	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000

Table 7.4: Two-channel (2×1) edge-case 1 million events counted per active channel. Sorted by active channels and phase patterns. 'A' is the first channel (ch0 for No Arbiter/FPA Cell, ch2 for RRA), and 'B' is the second channel (ch1 for No Arbiter/FPA Cell, ch3 for RRA). The phase patterns are offset by one clock (10 ns). Shaded cells have both channels arriving at the same clock cycle.

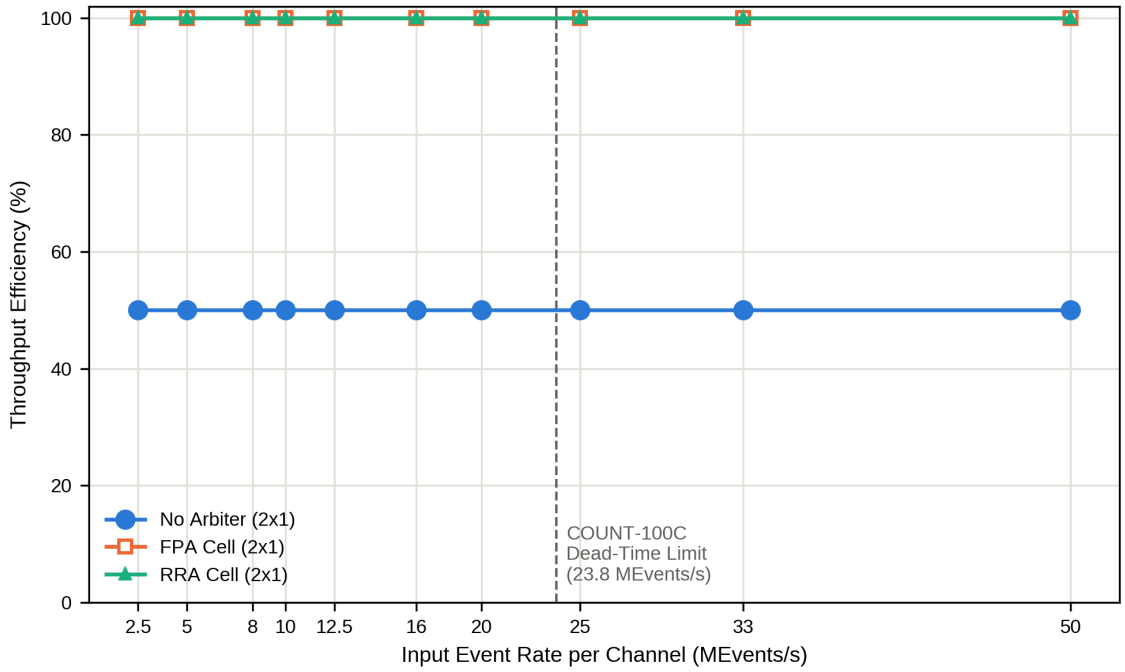


Figure 7.3: Two-channel throughput efficiency versus input event rate per channel. The dashed line marks the COUNT-100C dead time (23.8 MEvents/s).

The period sweep tests how each arbiter responds to an increase of event rate while under full contention (both channels active). All channels had a pulse width of 10 ns and a max of 1 million event signals per channel. Figure 7.3 plots the events lost with each design at different event rates. As discussed earlier, the no arbiter design would lose 50% of its events during full contention. But both the FPA and RRA cells were able to handle same time arrival. Results prove arbitration is required during same time events.

7.2.2 Four-Channel: No Arbiter vs FPA Tree vs RRA Tree

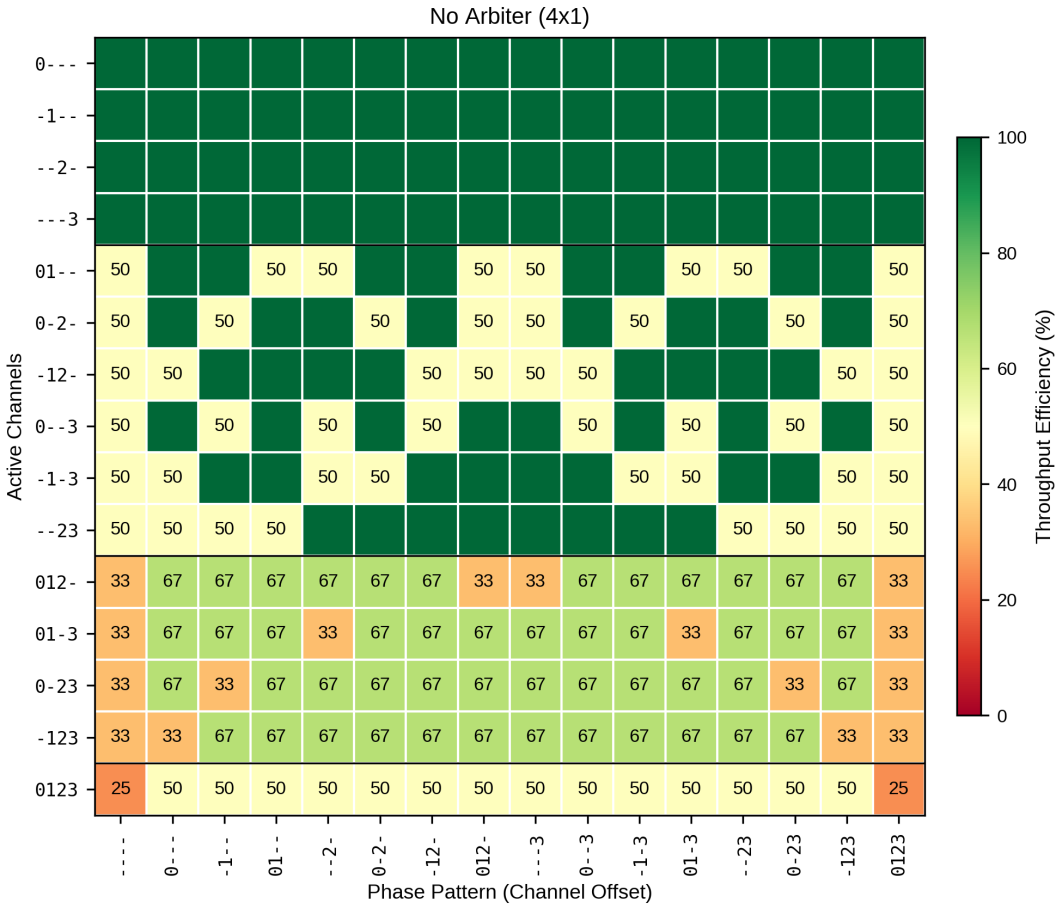


Figure 7.4: Four-channel throughput efficiency, No Arbiter (4×1) (period = **30 ns**, 1 million events per channel).

As expected from Figure 7.4, having no arbiter here shows events lost during contention between channels. This test used 1 million events per channel, period of 30 ns, width of 10 ns, and 240 combinations of active/inactive channels and various phases. With two channels active, half of the edge cases show loss. With three channels active, all edge cases experience loss. And with four channels active, this design experiences the worst case of loss with only 25% throughput. These results show worse affects of not having arbitration than Figures 7.1 and 7.2 proving further the critical need for arbitration to resolve these limitations.

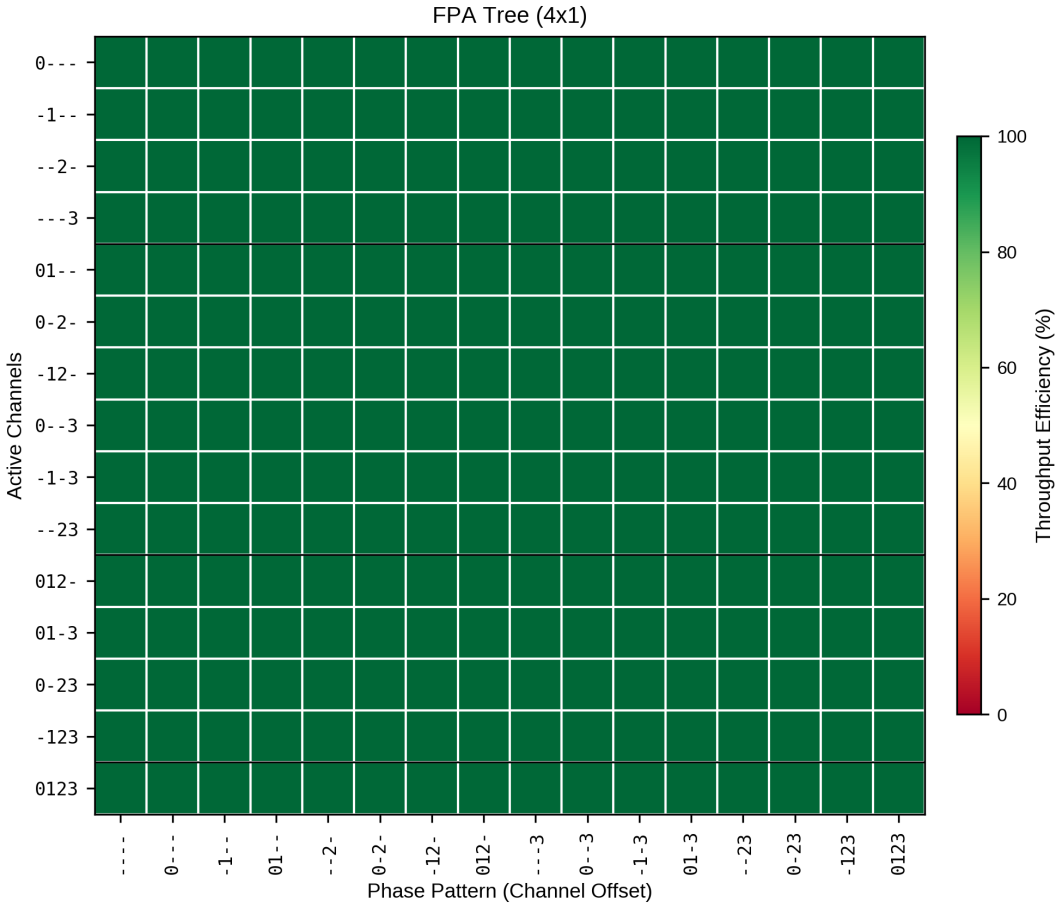


Figure 7.5: Four-channel throughput efficiency, FPA Tree (4×1) (period = **30 ns**, 1M events per channel).

Figure 7.5 shows the FPA Tree under the same conditions (1 million events per channel, period of 30 ns, width of 10 ns). The FPA tree was able to conserve all events across the 240 masks and phase combinations. This includes the cases that the no arbiter design lost events, where simultaneous arrivals engulfed each others signals. The arbiter tree serializes these contending requests allowing for multiple counts, rather than dropping them entirely. The request that loses during contention is held and granted only after the winner completes its handshake. At a 30 ns period, the tree is able to handle even a four-way collision before the next event arrives. However, throughput efficiency only measures whether the events were successfully counted, not which channel was served first. Below the FPA trees saturation, the arbitration policy shows no affect on the events counted. Its

affects only appear during saturation as shown in Figure 7.8 further below.

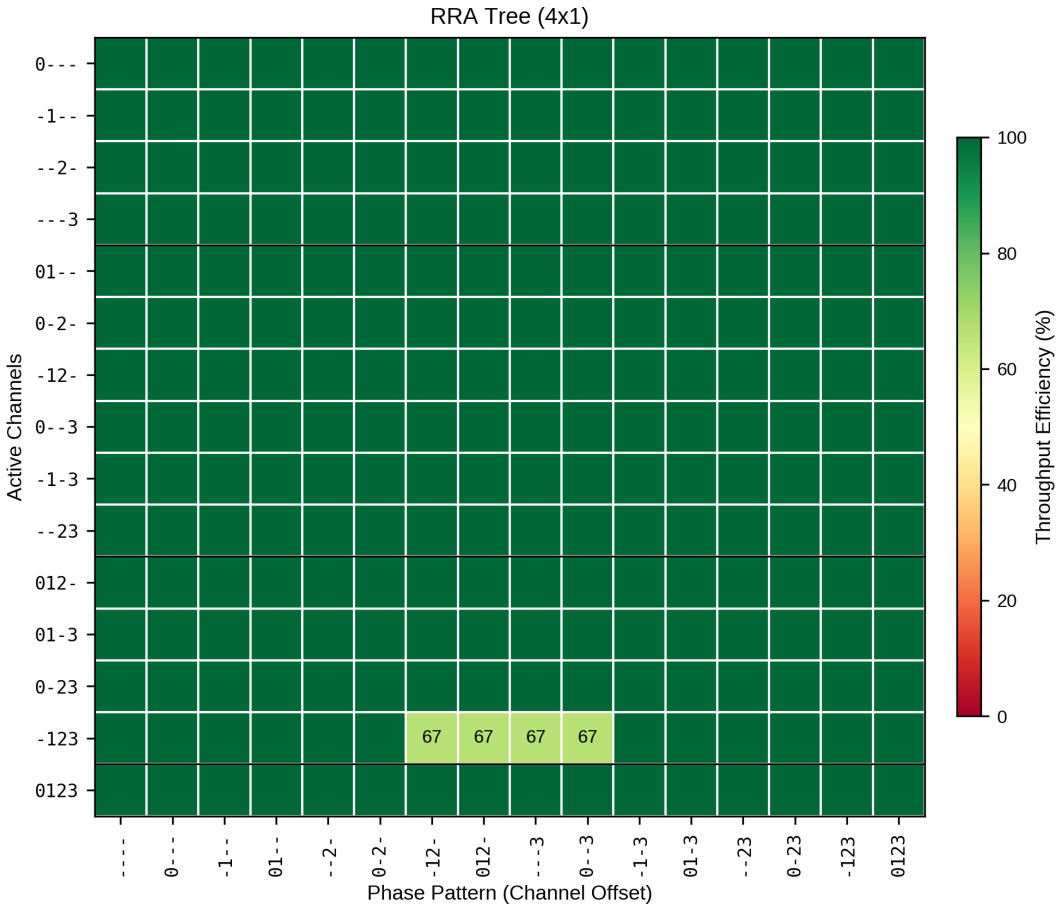


Figure 7.6: Four-channel throughput efficiency, RRA Tree (4×1) (period = **30 ns**, 1M events per channel).

RRA Tree (4×1) Failures – 30 ns Period

		Events counted per channel					
Active	Phase	Ch0	Ch1	Ch2	Ch3	Total	Lost
-123	-12-	–	1	1,999,998	1	2,000,000	1,000,000
-123	012-	–	0	1,999,998	2	2,000,000	1,000,000
-123	---3	–	0	1,999,999	1	2,000,000	1,000,000
-123	0--3	–	0	1,999,999	1	2,000,000	1,000,000

Table 7.5: RRA Tree (4×1) failed edge cases (period 30 ns, 1M events per channel. Shaded cell means a difference of more than 1% of what was offered).
(see complete counts in Appendix B).

Lastly, Figure 7.6 shows the RRA Tree under the same conditions as well (1 million

events per channel, period of 30 ns, width of 10 ns). The RRA tree, however, conserved every event in 236 edge cases out of the 240 combinations. Specifically, when channel 0 receives no new events, channels 1 and 2 arrive simultaneously, and channel 3 comes either before or after their contention. In these cases, channels 1 and 3 lose all of their events, while channel 2 records 2×10^6 events, twice the number it was offered. The tree therefore does more than drop events: it credits two of every three arrivals to channel 2. Because channels 1 and 2 are in different cells before it reaches the root, this would suggest the root cell fails to resolve the contention. However, when only channels 1 and 2 are active, the contention is successfully resolved. And when channels 0, 1, and 2 are active where channel 0 arrives either before or after the contention, the contention is again successfully resolved. This points to the failure of both channels 2 and 3's RRA cell alongside the root cell when contending with channels 0 and 1's cell at the root level. It should be noted that even during tests of longer periods and width, the RRA tree was unable to resolve this issue. So, the loss does not depend on event rate, rather it is a limitation in the RRA tree design.

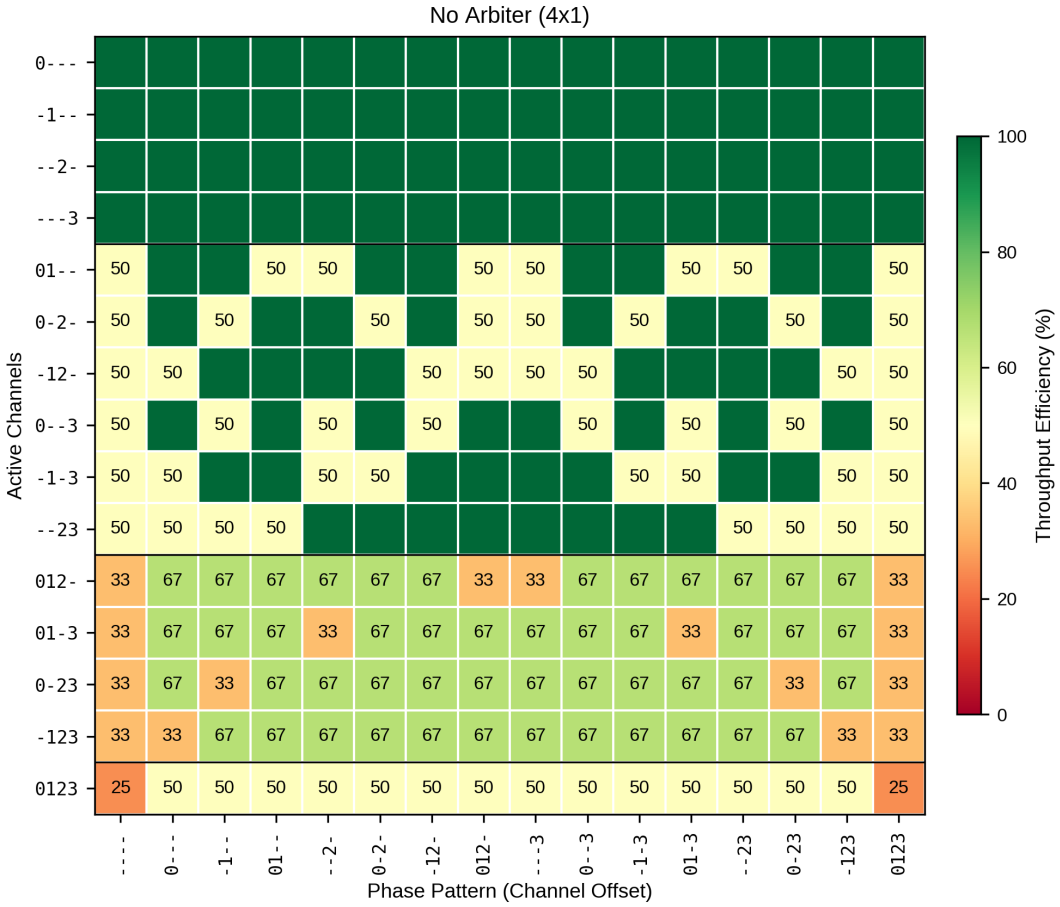


Figure 7.7: Four-channel throughput efficiency, No Arbiter (4×1) (period = **20 ns**, 1 million events per channel).

Figure 7.7 repeats the test of Figure 7.4 with event period reduced to 20 ns. The results are identical. There's no difference between the two results with the change of arrival time. This is expected, as the only complexity within this design is a 4-input OR-gate, merging the four channels. Because of that, we can expect event input speeds to not have as much of an impact as the FPA and RRA trees.

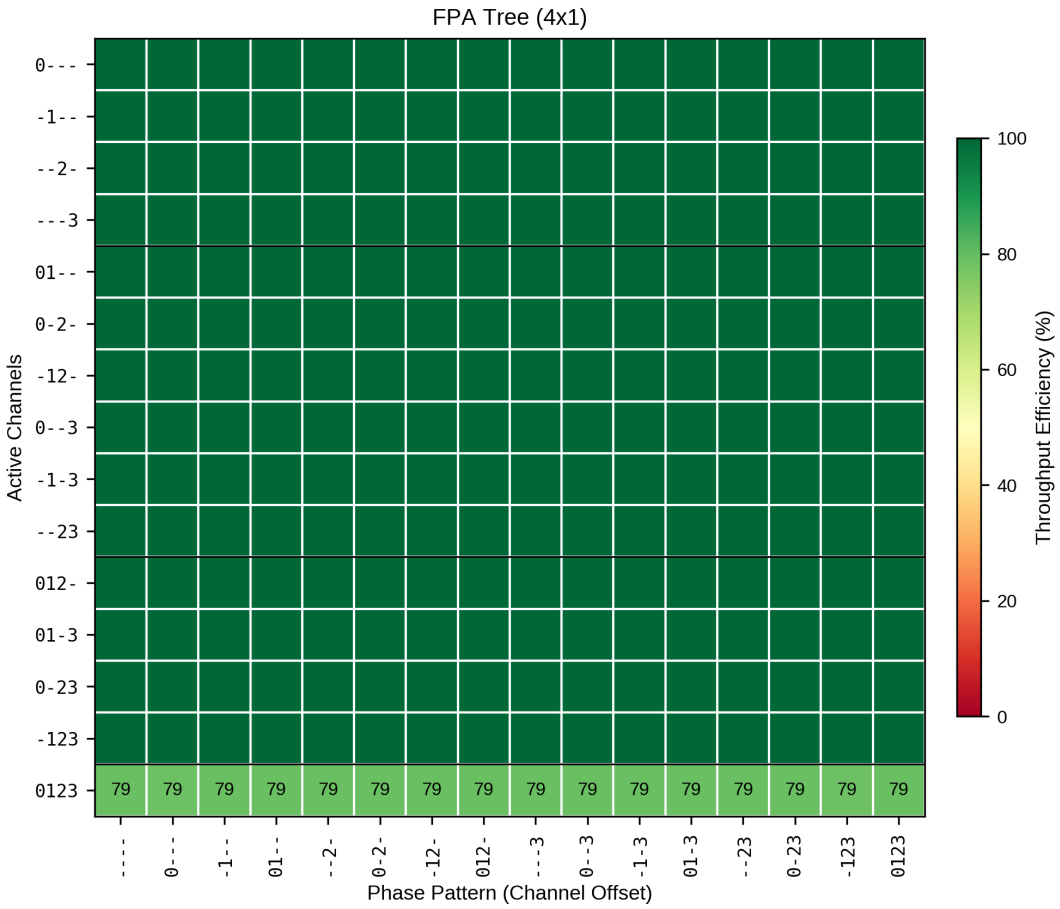


Figure 7.8: Four-channel throughput efficiency, FPA Tree (4×1) (period = **20 ns**, 1 million events per channel).

FPA Tree (4×1) Failures – 20 ns Period

		Events counted per channel					
Active	Phase	Ch0	Ch1	Ch2	Ch3	Total	Lost
0123	----	1,000,199	1,000,003	671,338	493,606	3,165,146	834,854
0123	0---	1,001,670	1,000,276	625,827	527,379	3,155,152	844,848
0123	-1--	1,001,684	1,000,281	632,382	524,701	3,159,048	840,952
0123	01--	1,000,195	999,998	661,965	503,102	3,165,260	834,740
0123	--2-	1,000,212	1,000,000	582,577	582,459	3,165,248	834,752
0123	0-2-	1,001,603	1,000,317	578,650	577,395	3,157,965	842,035
0123	-12-	1,001,577	1,000,306	578,873	577,335	3,158,091	841,909
0123	012-	1,000,179	1,000,000	669,699	495,239	3,165,117	834,883
0123	---3	1,000,216	999,999	670,341	494,638	3,165,194	834,806
0123	0--3	1,001,638	1,000,307	578,572	577,172	3,157,689	842,311
0123	-1-3	1,001,623	1,000,311	578,869	577,527	3,158,330	841,670
0123	01-3	1,000,236	1,000,002	582,521	582,488	3,165,247	834,753
0123	--23	1,000,257	1,000,002	661,236	503,789	3,165,284	834,716
0123	0-23	1,001,658	1,000,259	629,922	527,562	3,159,401	840,599
0123	-123	1,001,785	1,000,275	624,958	528,222	3,155,240	844,760
0123	0123	1,000,206	1,000,001	671,748	493,365	3,165,320	834,680

Table 7.6: FPA Tree (4×1) failed edge cases (period **20 ns**, 1M events per channel. Shaded cell means a difference of more than 1% of what was offered).
(see complete counts in Appendix B).

As mentioned previously, in Figure 7.8, we can see how event rate affects the throughput efficiency as the FPA tree enters saturation. This causes event loss shown in the last row, where all channels are active with varying phase shift patterns. Table 7.6 lists the per-channel counts for these cases. Under full contention, the arbiter applies a strict sequential priority policy (channel 0, 1, 2, then 3). Channels 0 and 1 record their full 1,000,000 events in every case, while all of the loss falls on channels 2 and 3, and channel 3 never records more events than channel 2. This illustrates the starvation risk inherent to a fixed priority design. Although, channel 3 is not outright starved, it is receiving fewer grants in every case. When requests arrive to channels 2 and 3 arrive simultaneously, channel 2 wins their shared FPA cell and records between 625,000 and 672,000 events, while channel 3 records between 493,000 and 528,000 events. When requests arrive phase shifted, they no longer contend within their shared FPA cell, and their counts get closer to matching. One exception to this is when channel 3 alone is offset from the other channels, (`---3` and `012-`), where channel

2 again records roughly 175,000 more events. This may be due to the contention between channels 0 and 1, and then the contention that would happen at the root cell, where channel 2 waits for channels 0 and 1 to clear their handshake.

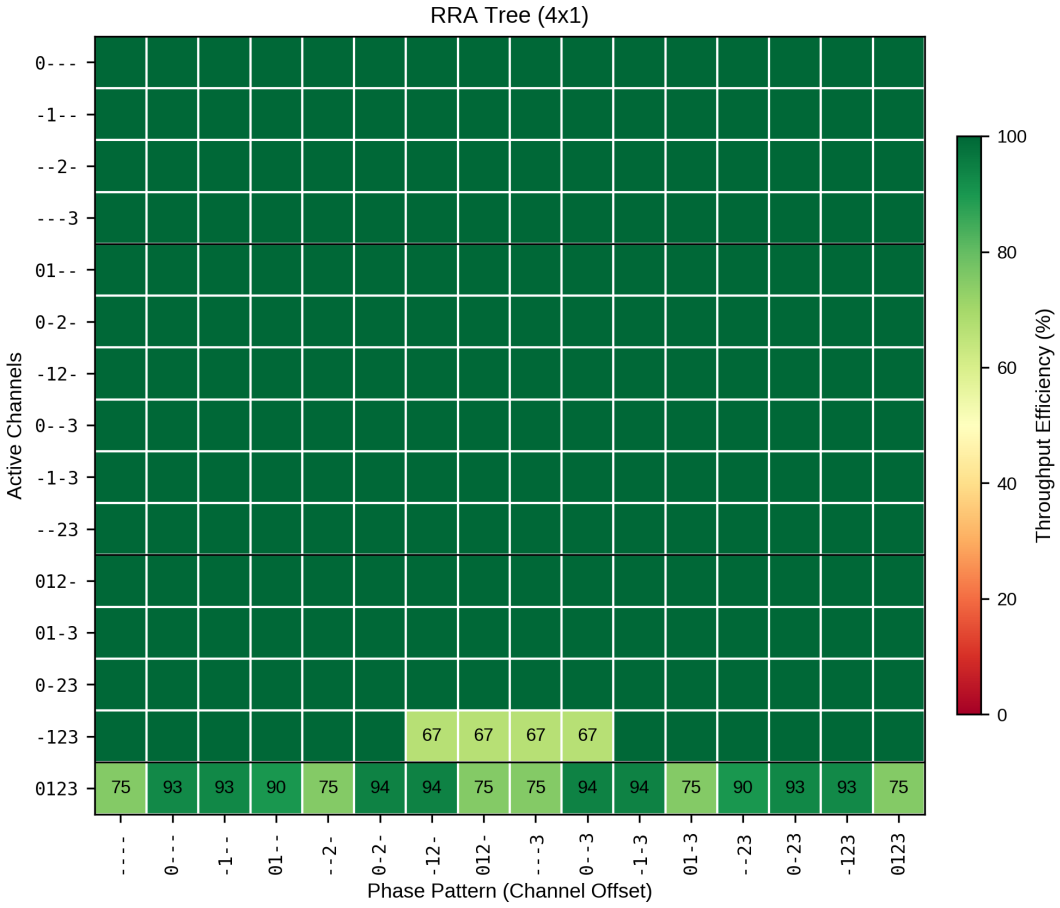


Figure 7.9: Four-channel throughput efficiency, RRA Tree (4×1) (period = **20 ns**, 1 million events per channel).

RRA Tree (4×1) Failures – 20 ns Period

		Events counted per channel					
Active	Phase	Ch0	Ch1	Ch2	Ch3	Total	Lost
-123	-12-	–	1	1,999,998	1	2,000,000	1,000,000
-123	012-	–	1	1,999,998	1	2,000,000	1,000,000
-123	---3	–	1	2,000,000	0	2,000,001	999,999
-123	0--3	–	2	1,999,999	0	2,000,001	999,999
0123	----	2	1,000,000	999,999	1,000,000	3,000,001	999,999
0123	0---	999,999	1,000,001	715,227	1,000,001	3,715,228	284,772
0123	-1--	1,000,000	1,000,000	720,837	1,000,001	3,720,838	279,162
0123	01--	999,998	1,000,000	600,003	1,000,000	3,600,001	399,999
0123	--2-	2	1,000,000	999,999	1,000,000	3,000,001	999,999
0123	0-2-	944,791	944,793	944,792	944,794	3,779,170	220,830
0123	-12-	944,678	944,679	944,678	944,679	3,778,714	221,286
0123	012-	6	999,999	999,999	1,000,001	3,000,005	999,995
0123	---3	2	1,000,000	999,999	1,000,000	3,000,001	999,999
0123	0--3	944,697	944,699	944,698	944,698	3,778,792	221,208
0123	-1-3	944,648	944,648	944,649	944,649	3,778,594	221,406
0123	01-3	7	999,999	1,000,000	1,000,000	3,000,006	999,994
0123	--23	1,000,000	1,000,000	600,000	1,000,000	3,600,000	400,000
0123	0-23	999,999	1,000,001	720,455	1,000,001	3,720,456	279,544
0123	-123	1,000,000	1,000,000	715,401	1,000,001	3,715,402	284,598
0123	0123	2	1,000,000	999,999	1,000,000	3,000,001	999,999

Table 7.7: RRA Tree (4×1) failed edge cases (period 20 ns, 1M events per channel. Shaded cell means a difference of more than 1% of what was offered).
(see complete counts in Appendix B).

At a 20 ns period, Figure 7.9 shows the RRA Tree failing 20 of the 240 cases, split between two rows: the active channels row `-123` failures carried over from the 30 ns results, and the active channel row `0123`, where the tree enters saturation. Like the FPA tree, the RRA tree only enters saturation once all channels are active. When contention occurs only at the root cell, rather than within the ch0/ch1 and ch2/ch3 leaves, the round-robin policy works as intended. Each channel records about 944,000 events. That is about a 94.5% throughput efficiency, surpassing the FPA tree at that edge case by about +15%. In the case where channel 2 is penalized, counting only 600,000 to 721,000 events, every other channel receives their full 1,000,000 events each. And the worst case, channel 0 is starved while the others receive their full 1,000,000 events each. These patterns point to how the RRA tree is implemented in fabric, not to the round-robin policy.

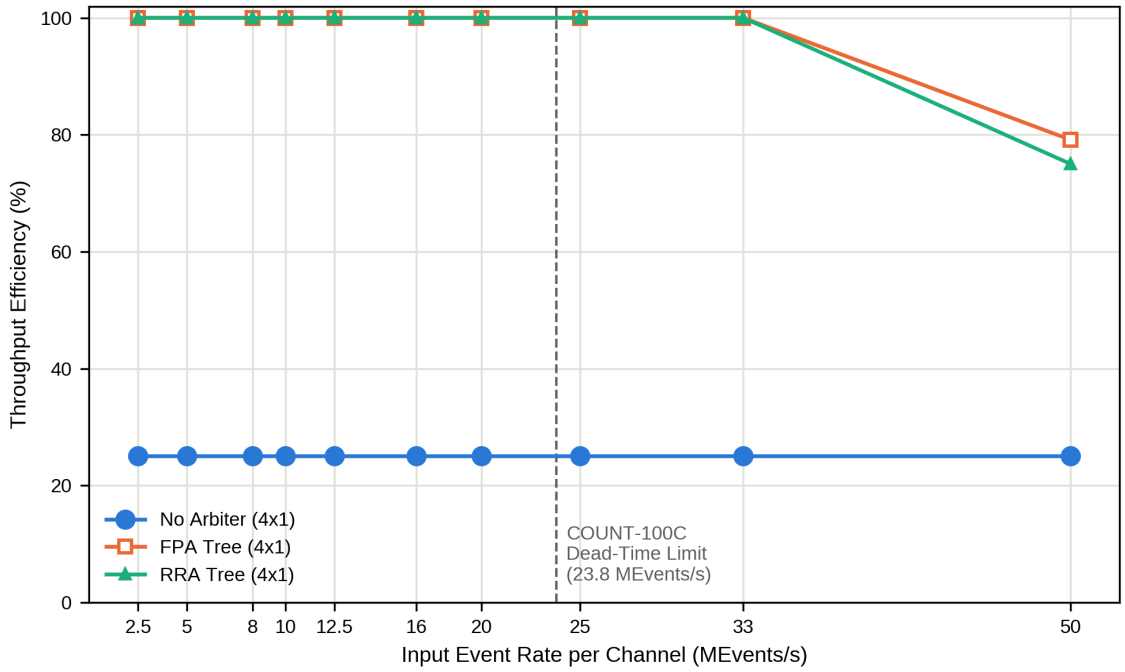


Figure 7.10: Four-channel throughput efficiency versus input event rate per channel. The dashed line marks the COUNT-100C dead-time limit (23.8 MEvents/s).

Figure 7.10 repeats the period sweep under the same conditions this time with the four-channel (4×1) designs. All 1 million events arrive at the same clock cycle (worst-case), and the throughput efficiency is calculated from the total events counted across all channels. Both trees conserved every event up to 33 MEvents/s per channel, but at 50 MEvents/s neither could keep up with the incoming requests. Throughput efficiency fell to 79.1% for the FPA Tree, and 75.0% for the RRA Tree, compared with 25% for the No Arbiter baseline design. The RRA tree's lower throughput efficiency is attributed to the channel 0 starvation edge case that was aforementioned. In 10 of the 16 edge cases with all four channels active, however, the RRA tree outperformed the FPA tree.

7.3 External Pulse Generators

The function generator drives all four channels, serving as an external signal source. Each burst emits a known number of pulses per channel, so every count has an exact ground truth. Uniquely, the function generator can output a specific frequency, permitting a relative

offset of less than one nanosecond between four independent channels. Because the machine uses the same clock, the offset is accurate. This allows us to see how the designs resolve imperfect signal requests. The function generator also allows us to strain test each channel individually for maximum throughput bandwidth till it reaches saturation. Because we already tested each channels max throughput efficiency while under full contention till it reached saturation.

7.3.1 Two-Channel: FPA Cell vs RRA Cell

Events lost	Edge offset (ns)								
Channel	0	0.25	0.5	0.75	1	1.5	2	3	5
ch0	14	15	15	16	15	16	16	16	16
ch1	15	15	15	16	15	16	16	16	16
Total	29	30	30	32	30	32	32	32	32

Table 7.8: FPA (2×1) Cell edge-offset sweep: events lost of 600,000 per channel.

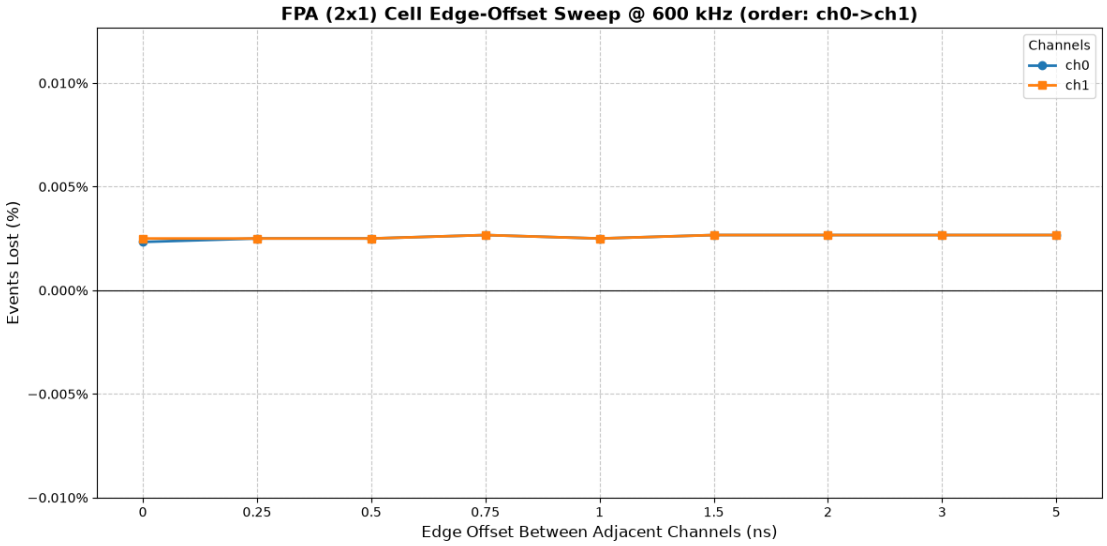


Figure 7.11: FPA (2x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator

Events lost	Edge offset (ns)								
Channel	0	0.25	0.5	0.75	1	1.5	2	3	5
ch2	15	40	15	16	15	16	16	16	16
ch3	15	17	15	16	15	16	16	16	16
Total	30	57	30	32	30	32	32	32	32

Table 7.9: RRA (2×1) Cell edge-offset sweep: events lost of 600,000 per channel.

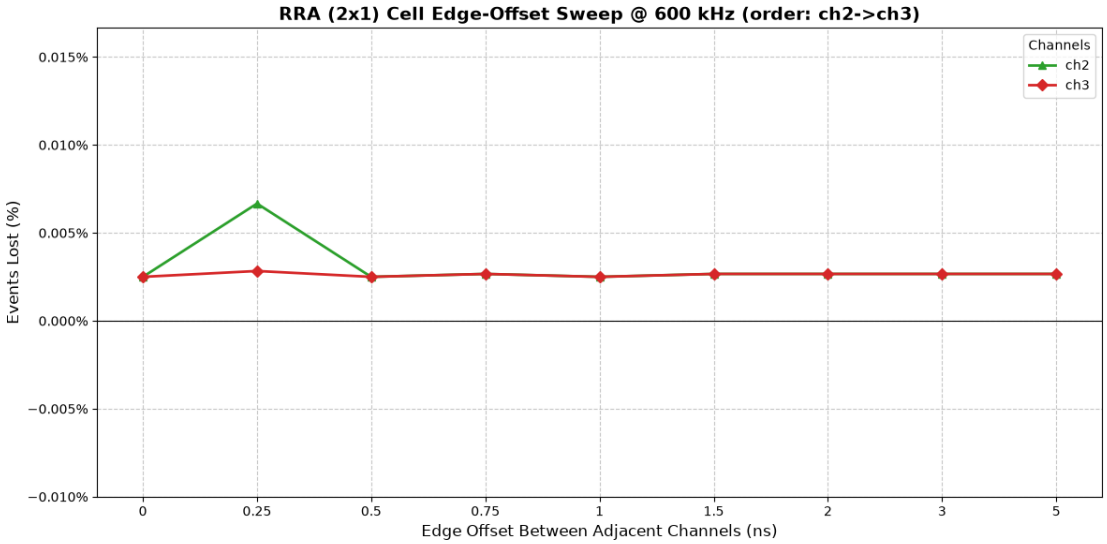


Figure 7.12: RRA (2x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator

$$\text{Events Lost } (\%) = \left(1 - \frac{N_{\text{counted}}}{N_{\text{expected}}}\right) \times 100\% \tag{7.4}$$

The actual amount of events to be counted was 600,000 because the pulse delay generator was outputting at a 600 kHz rate in a 1 second window for data collection. The actual amount of events taken gives us the results seen in Tables 7.8 and 7.9, visualized by the figures below them. With the smallest edge offset of 0.25 ns, the RRA cell ch2 lost 40 events, about 25 more than the 15–16 seen at every other offset. Despite the short offset, all other offset tests were relatively stable for both FPA and RRA cells.

FPA Cell (2×1) Single-Channel Rate Sweep

		Events Counted	
Rate (MHz)	Events Input	Ch0	Ch1
1	1,000,000	1,000,011	1,000,011
10	10,000,000	10,000,121	10,000,118
20	20,000,000	20,000,242	20,000,246
30	30,000,000	30,000,355	30,000,363
40	40,000,000	40,000,485	40,000,477
50	50,000,000	50,000,603	50,000,593
60	60,000,000	60,000,718	60,000,718
70	70,000,000	70,000,884	70,000,826
71	71,000,000	71,000,913	71,000,848
72	72,000,000	71,976,375	71,999,566
73	73,000,000	67,825,758	72,819,255
74	74,000,000	6,730,205	41,822,253
75	75,000,000	24,884	148,718
76	76,000,000	6	1,199
77	77,000,000	0	0
78	78,000,000	0	0

Table 7.10: FPA (2×1) Cell: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%.

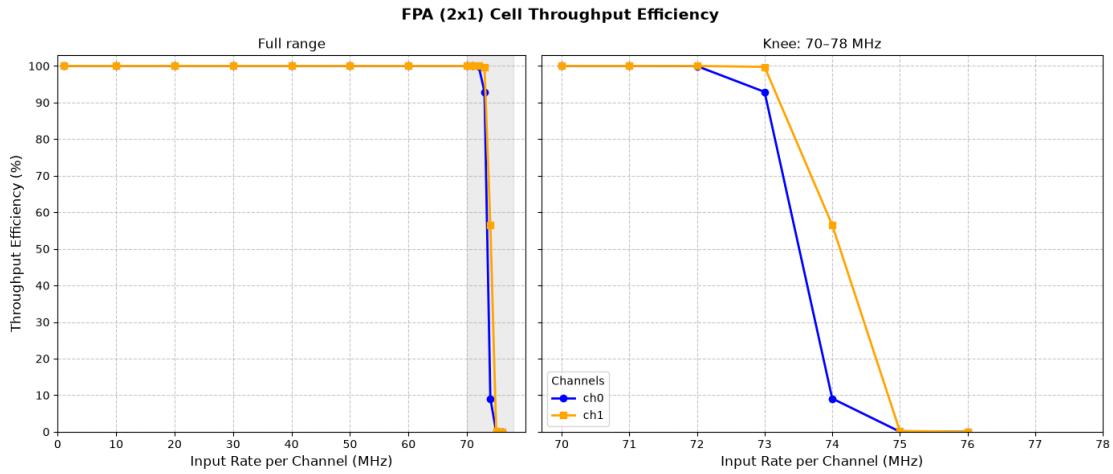


Figure 7.13: FPA (2×1) Cell: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events

RRA Cell (2×1) Single-Channel Rate Sweep

		Events Counted	
Rate (MHz)	Events Input	Ch2	Ch3
1	1,000,000	1,000,011	1,000,011
10	10,000,000	10,000,125	10,000,119
20	20,000,000	20,000,240	20,000,233
30	30,000,000	30,000,361	30,000,357
40	40,000,000	40,000,483	40,000,479
50	50,000,000	50,000,613	50,000,592
60	60,000,000	60,000,729	60,000,756
70	70,000,000	70,000,852	70,000,843
71	71,000,000	71,000,866	71,000,848
72	72,000,000	72,000,886	72,000,855
73	73,000,000	72,965,148	73,000,918
74	74,000,000	61,546,486	73,990,837
75	75,000,000	1,789,203	70,570,313
76	76,000,000	945	3,142,043
77	77,000,000	0	66
78	78,000,000	0	0

Table 7.11: RRA (2×1) Cell: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%.

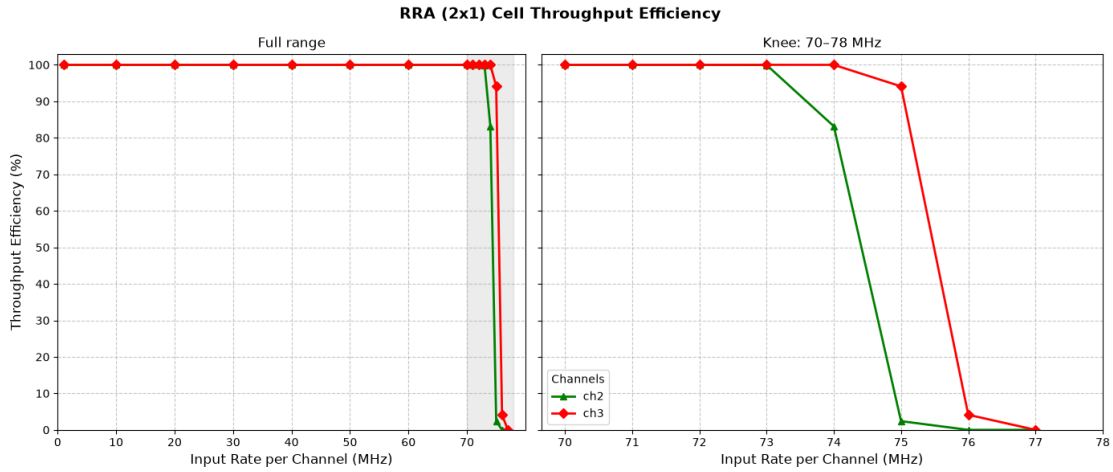


Figure 7.14: RRA (2×1) Cell: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events

Testing the FPA Cell (2×1) for single-channel throughput speed, as seen in Figure 7.13, a noticeable decline occurs at 73 MHz in ch0, then at 74 MHz ch1 follows. For the RRA Cell (2×1) show the decline starting at 74 MHz in ch2, then 75 MHz ch3 follows, as seen in Figure 7.14. This result tracks, as it takes less time in the RRA arbiter by

roughly 1.3 ns as discussed earlier in Table 7.2. What this means, on a single-channel with no contention, the four-phase handshake for the FPA cell develops issues at 73 MHz, while the RRA cell develops issues at 74 MHz. So the four-phase handshakes rising and falling transitions are not fully performed during the 73-74 MHz event rate [15]. Which suggests the request-acknowledge cycle is no longer keeping pace with the event rate.

7.3.2 Four-Channel: FPA Tree vs RRA Tree

Events lost	Edge offset (ns)								
Channel	0	0.25	0.5	0.75	1	1.5	2	3	5
ch0	−261	20	17	15	15	17	16	16	17
ch1	−568	12	17	15	15	17	16	16	17
ch2	3,305	16	17	16	15	17	16	16	17
ch3	18	15	17	17	15	17	16	16	17
Total	2,494	63	68	63	60	68	64	64	68

Table 7.12: FPA (4×1) Tree edge-offset sweep: events lost of 600,000 per channel. Negative values indicate more events counted than given.

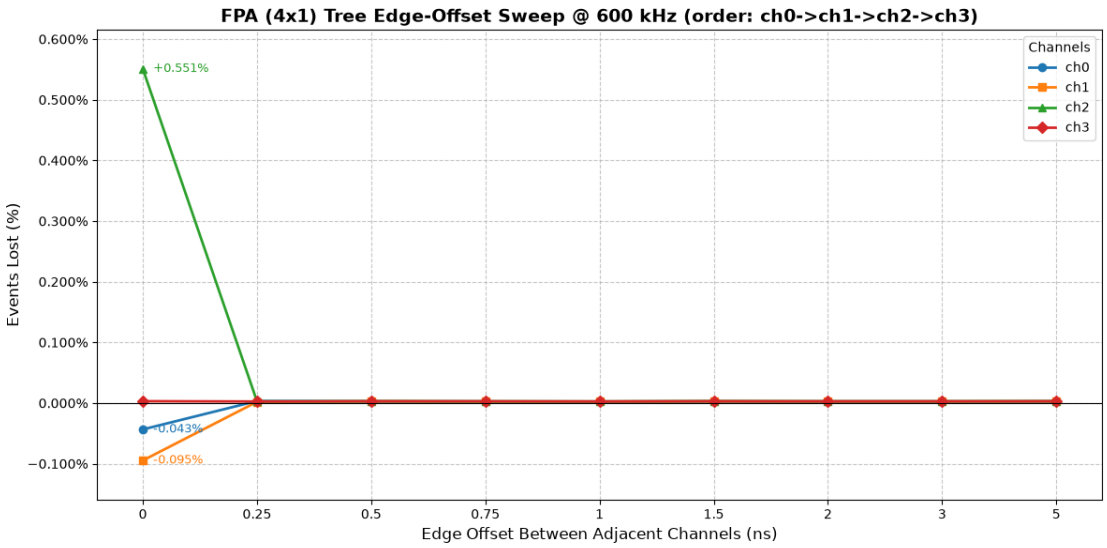


Figure 7.15: FPA (4x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator

Events lost	Edge offset (ns)								
Channel	0	0.25	0.5	0.75	1	1.5	2	3	5
ch0	15	17	15	15	15	16	15	16	16
ch1	15	17	15	15	15	16	15	16	16
ch2	15	17	15	15	15	16	15	16	16
ch3	15	17	15	15	15	16	15	16	16
Total	60	68	60	60	60	64	60	64	64

Table 7.13: RRA (4×1) Tree edge-offset sweep: events lost of 600,000 per channel.

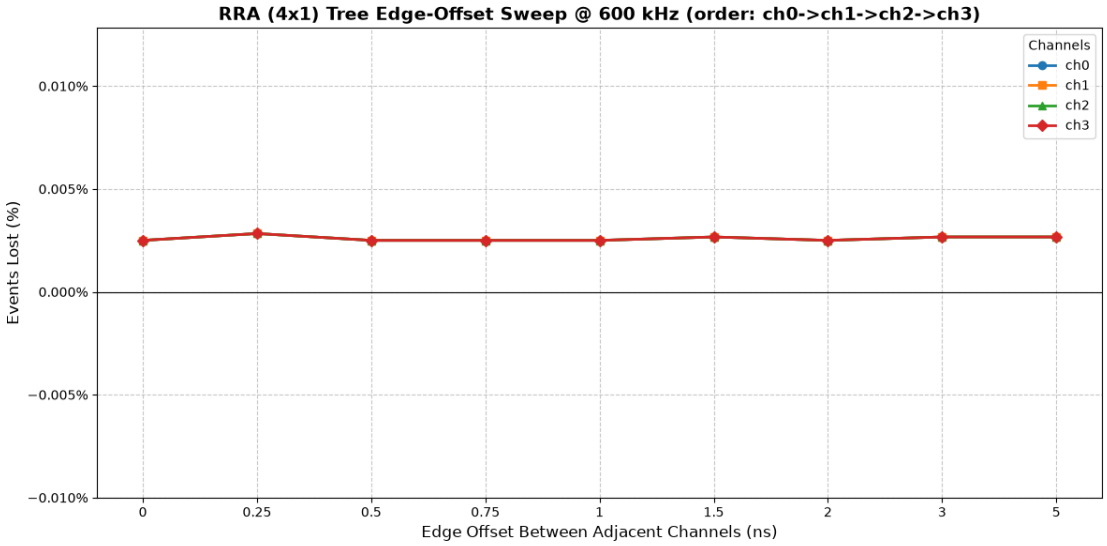


Figure 7.16: RRA (4x1) per-channel events lost vs. edge offset, 600 kHz Pulse Delay Generator

Just like the arbiter cells, the trees go through the same edge-offset sweep. The noticeable outlier is the 0 ns offset in Table 7.12. Unlike what is shown in the other offset results, an outlier this large at this point is abnormal. Unlike results previously shown in Figure 7.5, full contention succeeded at a faster event rate. This leads to questions on how the arbiters respond to slower event rates at full contention; another objective to tackle under future work. Otherwise, all things considered ch2 lost roughly 0.55% of its events, and the tree as a whole 0.10%. Every other data point seemed very stable with very minimum event lost rate.

FPA Tree (4×1) Single-Channel Rate Sweep

		Events Counted			
Rate (MHz)	Events Input	Ch0	Ch1	Ch2	Ch3
1	1,000,000	1,000,009	1,000,009	1,000,009	1,000,009
10	10,000,000	10,000,097	10,000,097	10,000,097	10,000,097
20	20,000,000	20,000,195	20,000,195	20,000,195	20,000,195
30	30,000,000	30,000,311	30,000,312	30,000,312	30,000,316
40	40,000,000	40,000,412	40,000,418	40,000,414	40,000,414
50	50,000,000	50,000,515	50,000,514	50,000,518	50,000,524
60	60,000,000	60,000,610	60,000,615	60,000,610	60,000,623
70	70,000,000	70,000,712	70,000,733	70,000,727	70,000,719
71	71,000,000	71,000,732	71,000,735	71,000,728	71,000,737
72	72,000,000	72,000,746	72,000,750	72,000,745	72,000,757
73	73,000,000	72,999,769	73,000,768	73,000,769	73,000,774
74	74,000,000	73,707,355	73,971,020	73,995,670	74,000,796
75	75,000,000	37,579,859	56,286,908	61,461,452	74,974,068
76	76,000,000	343,295	15,507,058	26,278,945	74,950,924
77	77,000,000	228	16,686	17,856	16,516,284
78	78,000,000	0	0	0	276

Table 7.14: FPA (4×1) Tree: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%.

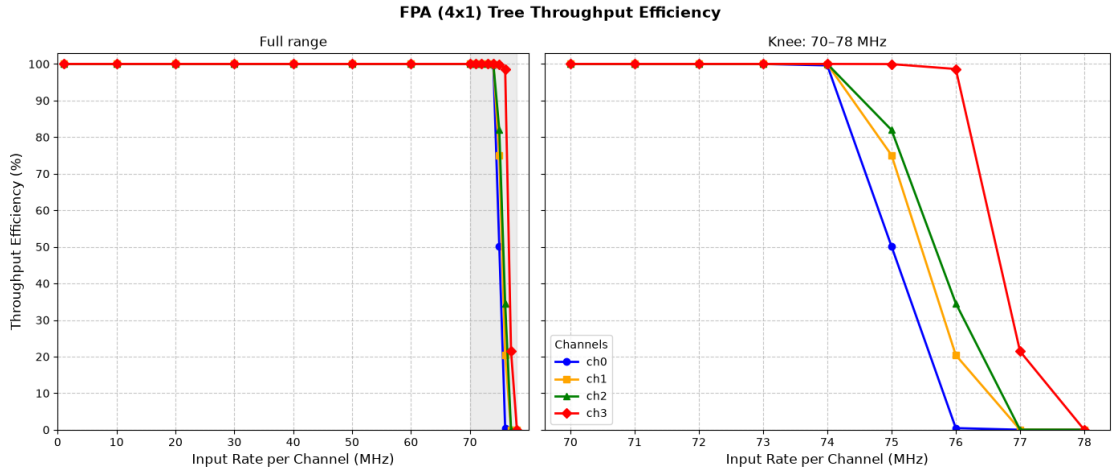


Figure 7.17: FPA (4×1) Tree: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events

RRA Tree (4×1) Single-Channel Rate Sweep

		Events Counted			
Rate (MHz)	Events Input	Ch0	Ch1	Ch2	Ch3
1	1,000,000	1,000,010	1,000,010	1,000,010	1,000,010
10	10,000,000	10,000,112	10,000,111	10,000,114	10,000,113
20	20,000,000	20,000,225	20,000,231	20,000,230	20,000,230
30	30,000,000	30,000,342	30,000,335	30,000,333	30,000,347
40	40,000,000	40,000,452	40,000,451	40,000,455	40,000,453
50	50,000,000	50,000,565	50,000,551	50,000,555	50,000,561
60	60,000,000	60,000,694	60,000,669	60,000,671	60,000,664
70	70,000,000	70,000,799	70,000,788	70,000,789	70,000,799
71	71,000,000	71,000,792	71,000,813	71,000,825	71,000,806
72	72,000,000	72,000,809	72,000,814	72,000,803	72,000,778
73	73,000,000	73,000,051	73,000,789	73,000,797	73,000,804
74	74,000,000	73,774,225	73,988,890	73,997,706	74,000,806
75	75,000,000	37,779,589	70,896,761	72,345,996	75,000,273
76	76,000,000	111,299	4,841,684	8,696,852	67,362,605
77	77,000,000	135	9,695	144	2,397,054
78	78,000,000	0	0	0	759

Table 7.15: RRA (4×1) Tree: Total events counted per channel from the function generator. Shaded cells differ from the events input by >1%.

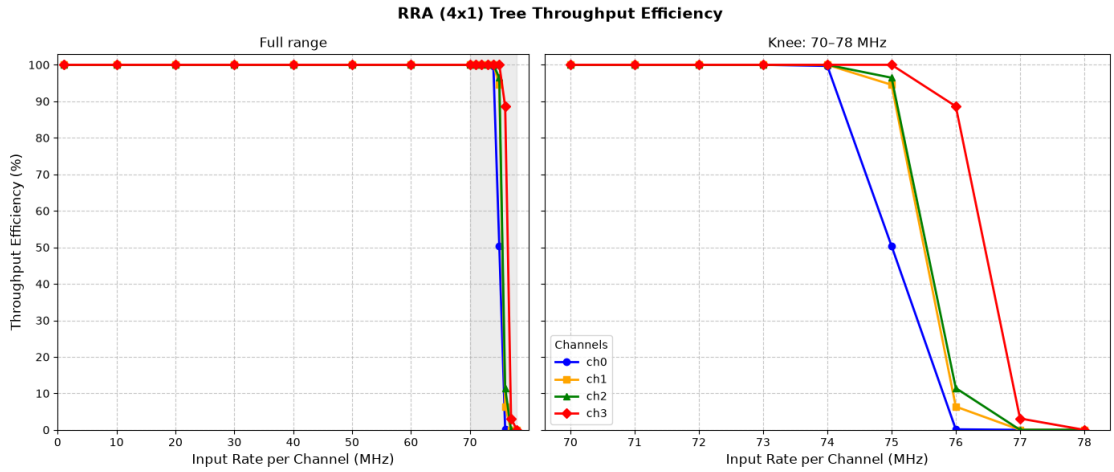


Figure 7.18: RRA (4×1) Tree: Per-channel throughput efficiency (%) vs. input rate (MHz), $f \times 10^6$ events

This time around, the FPA and RRA trees start saturating around 74 MHz, which is roughly the same throughput efficiency drop off point that was seen in FPA and RRA cells single-channel throughput test. This is a good sign, this means despite increasing the cells, the tree structure still maintained the throughput integrity. However, this now changes the

interpretation discussed in Section 7.3.1. Shown in Table 7.2, the FPA and RRA tree have longer total delay with a difference between cell to tree of 3 ns-4 ns (one-way), and the change of 1 MHz frequency when the events are lost is the equivalent of 0.5 ns. So, the expected result should have been noticeably worse. This means the arbiter is not the bottleneck, it is something further ahead of it. One possibility is the limits of the PMOD is that it is not meant to handle high-speeds [7]. This puts into question whether the high-speed SYZYGY connector ports could either solve this issue, or shine a new light on it. This is something that needs to be investigated in future work.

7.4 SPAD Detector

Initially, the reference rates were recorded on the commercial photon counters (SR400 / SR430) by going through ever test point, 0 µA to 100 µA current into the LED, before the FPGA measurements were taken. During the two-channel cell tests with the COUNT-100C SPAD, it was found that the path between the integrating sphere and the SPAD was found to admit ambient light. This was observed mid test when the count rate varied noticeably as objects moved around the detector. This indicates that part of the measured rate includes stray light, and not just the LED illumination from the integrating sphere. This discovery occurred after testing the arbiter trees, seen in Section 7.4.2. Instead of removing the results and redoing the test, the cells were tested next using a different recording technique. So results will look significantly different when comparing Sections 7.4.1 and 7.4.2. The new recording technique used was to record the photon counts on the FPGA, reconnect to the SR400 then record the photon counts, and then increment the current source to the next point. This closes the gap between testing the same point and get us closer to a more accurate read comparison.

7.4.1 Two-Channel: FPA Cell vs RRA Cell

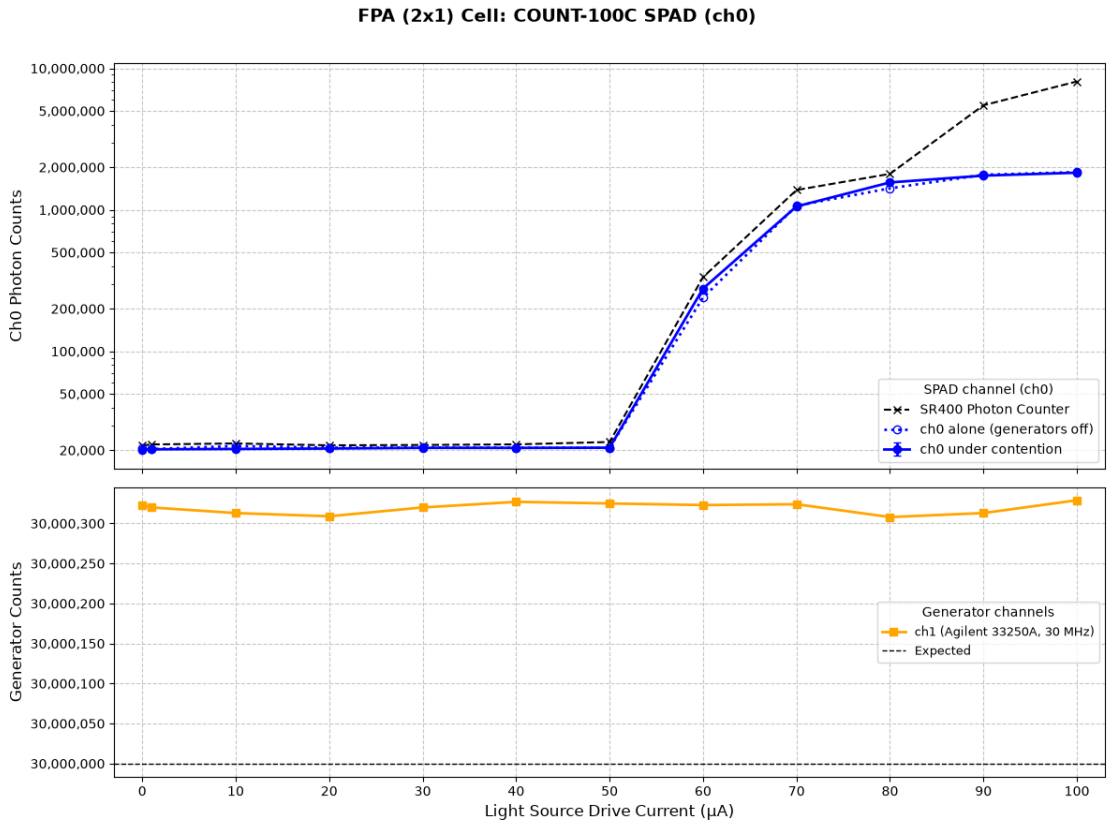


Figure 7.19: FPA (2×1) Cell: ch0 SPAD count rate (cps) vs. light source drive current (µA), SR400 and FPGA comparison, ch1 at 30 MHz by the function generator.

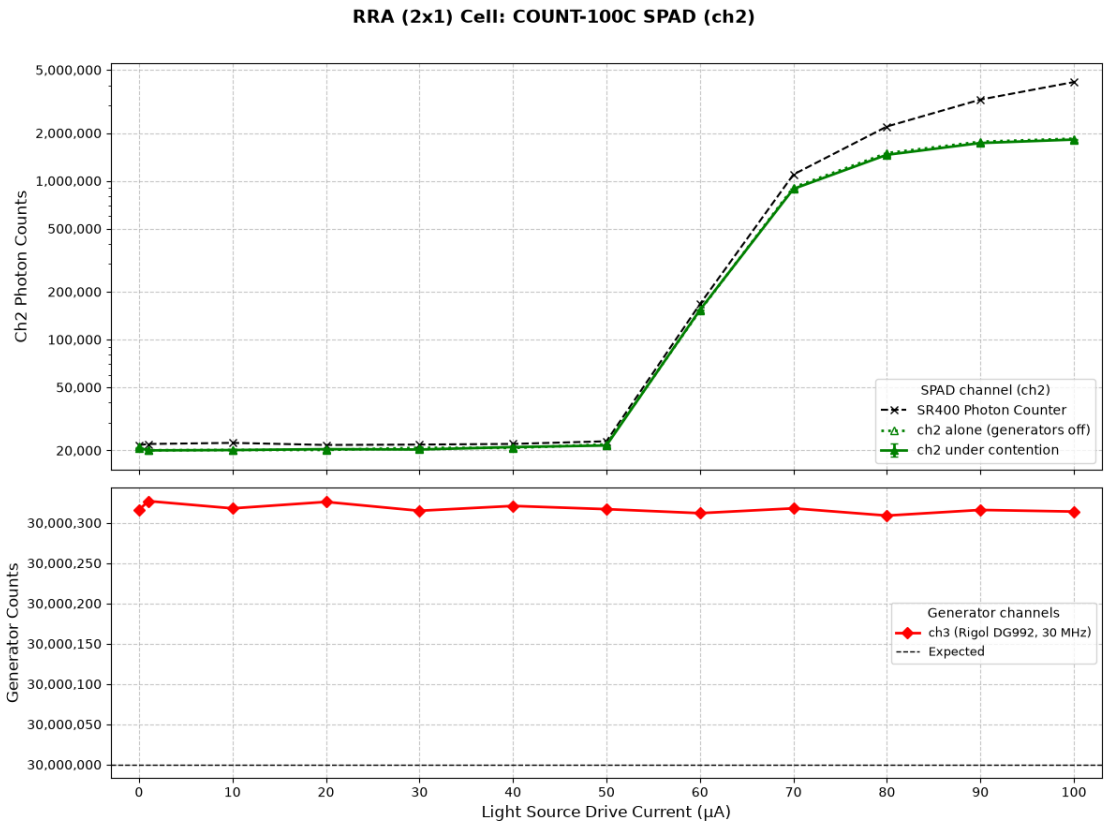


Figure 7.20: RRA (2×1) Cell: ch2 SPAD count rate (cps) vs. light source drive current (µA), SR400 and FPGA comparison, ch3 at 30 MHz by the function generator.

By switching the recording technique as mentioned earlier, the gap between the commercial photon counter and the FPGA closed in when comparing it to both arbiter tree results below, as shown in Figures 7.19 and 7.20. Although the photon counts begin to hedge off at the 80 µA point, the number of photons counted between the two stayed relatively close, between 0 µA to 60 µA. For the FPA, between 0 µA to 50 µA the count difference averaged a 5.4% difference, where the SR400 counted between 800-2,000 more photons compared to the FPGA. For the RRA, between 0 µA to 70 µA the count difference averaged a 5.25% difference, where the SR400 counted between 400-2,000 more photons compared to the FPGA. Once the LED was driven to 60 µA and up, the SR400 pulls away and the FPGA starts to plateau at around 70 µA.

7.4.2 Four-Channel: FPA Tree vs RRA Tree

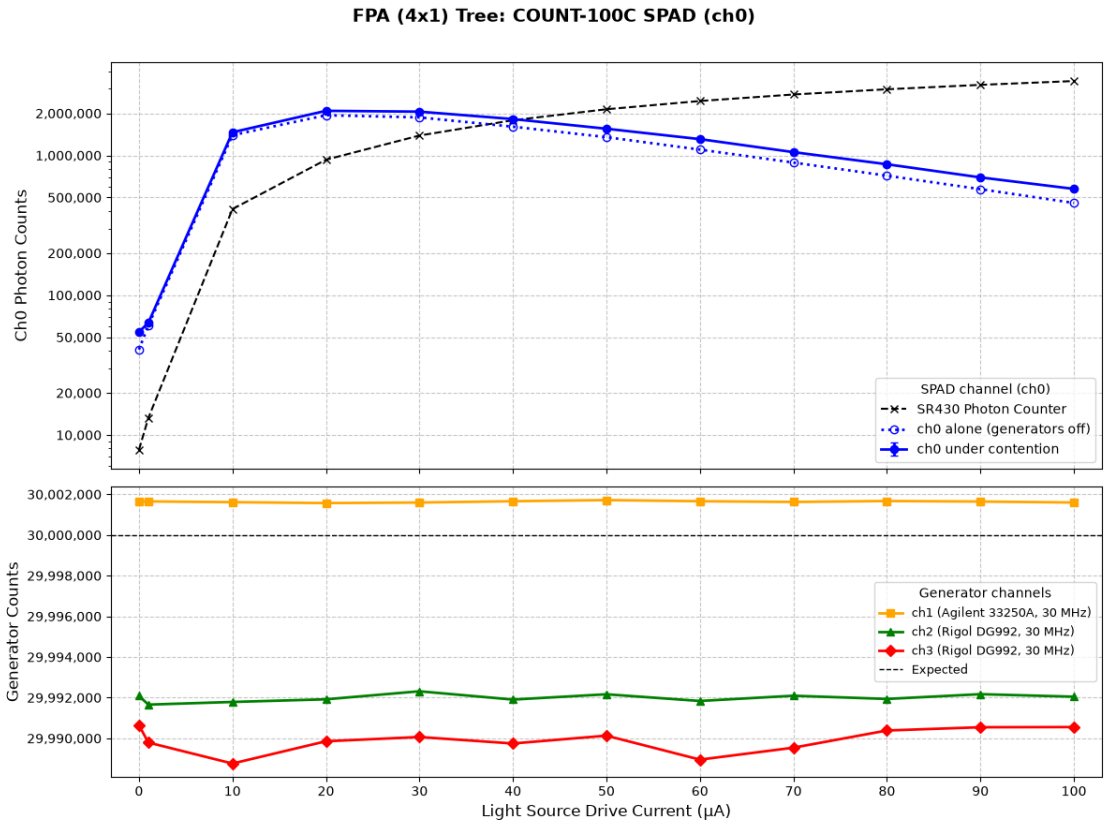


Figure 7.21: FPA (4×1) Tree: ch0 SPAD count rate (cps) vs. light source drive current (µA), SR430 and FPGA comparison, ch1/ch2/ch3 at 30 MHz by the function generators.

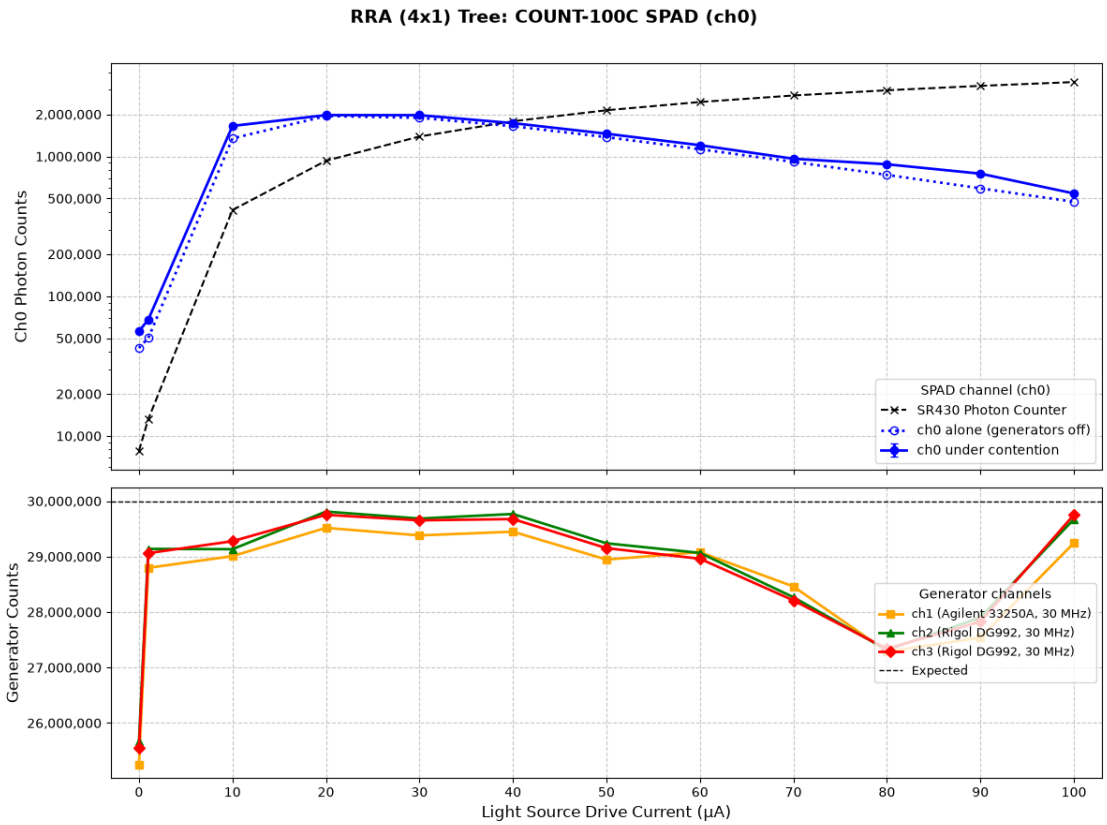


Figure 7.22: RRA (4×1) Tree: ch0 SPAD count rate (cps) vs. light source drive current (µA), SR430 and FPGA comparison, ch1/ch2/ch3 at 30 MHz by the function generators.

The photon counting setup, built from Thorlabs, was assumed to fully enclose the light path from the LED to the SPAD. This assumption proved incorrect, as it was discovered mid-test with large objects hovering the photon counting setup it can influence the counts in both the SR430 and the FPGA. Because the SR430 counts were recorded for all test points before the FPGA measurements were made, the light leak contribution differs between the two. So, a direct comparison between Figures 7.21 and 7.22 is unreliable. However, both trees were still able to produce near identical count curves when comparing the two. The FPA tree was able to conserve its generated counts to within 0.02%, while the RRA tree by contrast lost 1% to 15% of its generated events. The drop in generator counts may be attributed to the issues seen in Figure 7.6.

Chapter 8: Conclusions and Future Work

8.1 Conclusion

After going through rigorous testing during both implementation and evaluation, both arbiter cells and trees showed promising results. Both arbiters counted events to within 0.002% up to 33 MEvents/s per channel (a 30 ns period), which is well above the COUNT-100C's 23.8 MEvents/s dead time limit, with the exception of four edge cases from the RRA tree. The RRA tree's edge case failures shown in Figure 7.6 also appeared during the SPAD count test seen in Figure 7.22. These failures could be solved by exploring placement delay solutions as shown in Table 7.2. So, using asynchronous arbiters on an FPGA needs more hardware validation because simulation and design alone can not fully predict its behavior. It is also worth noting that while below saturation, both arbiters we able to conserve every event offered. However, at saturation, the FPA tree's strict priority showed signs of channel starvation (ch3 down to 49% events counted), while the RRA tree's fair priority showed each channel at 94.5% (see Tables 7.6 and 7.7). So, for photon counting at high speeds and high chances of full channel contention, the RRA implementation is the better policy. But, the specific RRA implementation used in this thesis needs the edge case fixed beforehand. With all that being said, asynchronous arbitration can be implemented in commercial FPGA fabric for photon counting. The risk is in how the board implements the arbiters post-implementation. A thorough hardware validation must be conducted.

8.2 Future Work

Some issues discovered during this research showed room for improvement that could better implement and evaluate asynchronous arbiters on an FPGA board. These issues are re-highlighted below, alongside the potential solution for them.

- Abnormal behavior shown in the FPA tree at full contention with a slow event rate (see Figure 7.15). A solution to it could be bundling the `data_out` origin with the request, that way no requests will be absorbed into other channels. This is seen in Sparsø's 4-phase dual-rail protocol [15].
- The single-channel throughput ceiling identified in Figures 7.17 and 7.18. A solution to this could be using a high-speed input port like the SYZYGY (Zmod) port on the Eclypse Z7 which is rated for high-speed frequencies [7]. As an example, high-speed Zmod modules are already supported on the board [5].
- The asynchronous to synchronous readout hazard discussed in Section 3.4. Implementing a register before the gray code synchronizes via CDC [8].
- Placement delays seen throughout all arbiter designs (discussed in Section 7.1). To control placement delay, use LUT instantiations to create the logic to be able to use placement constraints on them. This allows the synthesis to map the arbiter logic manually in the constraints.
- In both the cell and tree data sets taken with the photon counter (Section 7.4), the FPGA count diverges from the commercial photon counter reference. While the SR400/SR430 keeps rising with drive current, the FPGA count saturates sooner and, in the trees, declines. Because the SPAD-channel-alone curves coincide with the contention curves in Figures 7.19, 7.20, 7.21, and 7.22, the arbiter is not the cause. The difference therefore lies upstream of the arbiter, between the FPGA input path and the discriminator front ends of the commercial instruments, and further investigation is needed to establish what the two counters can be directly compared on.

Appendix A: Verilog Source Code

A.1 Asynchronous Flip-Flop

```
`timescale 1ns/1ps

module apd_req_latch (
    input  wire resetn,

    input  wire pulse,      // raw APD
    input  wire ack,        // async clear
    output wire req         // resampled
);

    (* keep = "true", dont_touch = "true" *) reg req_r = 1'b0;

    wire clr = ~resetn | ack;

    always @(posedge pulse or posedge clr) begin
        if (clr) req_r <= 1'b0;
        else     req_r <= 1'b1;
    end

    assign req = req_r;

endmodule
```

A.2 No Arbiter

```
`timescale 1ns/1ps

module no_arbiter #(
    parameter N     = 4,
    parameter WIDTH = (N <= 2) ? 1 : $clog2(N)  // if N = 4, WIDTH = 2
) (
    input  wire             resetn,

    input  wire [N-1:0]     req,
    output wire [N-1:0]     ack,

    output wire             grant,
    output wire [WIDTH-1:0] origin,
    input  wire             grant_ack
);

```

```
    // OR gating all requests
    // any overlapping events merge into one grant
    assign grant = resetn & (|req);

    // on grant_ack, clear every requesting latch (merged events are lost)
    assign ack = {N{grant_ack}} & req;

    // lowest-index active channel
    reg [WIDTH-1:0] org_r;
    generate
        if (N == 4) begin : fourCh
            always @(*) begin
                if (req[0])
                    org_r = 2'b00;
                else if (req[1])
                    org_r = 2'b01;
                else if (req[2])
                    org_r = 2'b10;
                else if (req[3])
                    org_r = 2'b11;
                else
                    org_r = 2'b00;
            end
        end
        else if (N == 2) begin : twoCh
            always @(*) begin
                if (req[0])
                    org_r = 1'b0;
                else if (req[1])
                    org_r = 1'b1;
                else
                    org_r = 1'b0;
            end
        end
    endgenerate

    assign origin = org_r;

endmodule
```

A.3 FPA Cell (2×1)

```
`timescale 1ns / 1ps

module fpa_2to1 #(
    parameter WIDTH       = 1,      // width of the data bus (default = 1)
    parameter NUM_BUFFERS = 0
) (
    input  wire             rstn,    // Active LOW from USER

```

```
    // downstream face (toward detectors / child cells)
    input  wire              req0,    // ch0
    input  wire              req1,    // ch1
    input  wire [WIDTH-1:0]  data0,   // ch0 data bus
    input  wire [WIDTH-1:0]  data1,   // ch1 data bus
    output wire              ack0,    // active LOW
    output wire              ack1,    // active LOW

    // upstream face (toward readout / parent cell)
    output wire              fpa_req,
    output wire [WIDTH-1:0]  data_out,
    input  wire              ack_fpa  // active LOW
);

    (* dont_touch = "true" *) wire r0_in, r1_in;
    (* dont_touch = "true" *) wire a0, a1;

    // priority network
    wire r0 = rstn & req0; //(rstn & req0 & ~req1) | (rstn & req0 & req1);   // ch0 HIGH OR ch0/ch1 collision
    wire r1 = rstn & ~req0 & req1;                              // ch0 LOW AND ch1 HIGH

    // async latch
    assign r0_in = r0 & ~a1;
    assign r1_in = r1 & ~a0;
    muller_c_element c0 (.rstn(rstn), .A(r0_in), .B(ack_fpa), .Y(a0));
    muller_c_element c1 (.rstn(rstn), .A(r1_in), .B(ack_fpa), .Y(a1));
    assign ack0 = ~a0;
    assign ack1 = ~a1;

    wire pre_fpa_req = a0 | a1;
    buffer #(.NUM_BUFFERS(NUM_BUFFERS)) fpa_buffer (.in(pre_fpa_req), .out(fpa_req));

    // data bus (origin select)
    (* dont_touch = "true" *) wire [WIDTH-1:0] data = (a1) ? data1 : data0;
    data_latch #(.WIDTH(WIDTH)) d_latch (
        .rstn(rstn),             // reset
        .req(fpa_req), .D(data), // D-Latch inputs
        .Q(data_out)             // D-Latch output
    );

endmodule
```

A.4 RRA Cell (2×1)

```
`timescale 1ns / 1ps

module rra_2to1 #(
    parameter WIDTH       = 1,      // width of the data bus (default = 1)
    parameter NUM_BUFFERS = 0
```

```
) (
    input  wire              rstn,    // Active LOW from USER

    // downstream face (toward detectors / child cells)
    input  wire              req0,    // ch0
    input  wire              req1,    // ch1
    input  wire [WIDTH-1:0]  data0,   // ch0 data bus
    input  wire [WIDTH-1:0]  data1,   // ch1 data bus
    output wire              ack0,    // active LOW
    output wire              ack1,    // active LOW

    // upstream face (toward readout / parent cell)
    output wire              fpa_req,
    output wire [WIDTH-1:0]  data_out,
    input  wire              ack_fpa  // active LOW
);

    (* dont_touch = "true" *) wire r0_in, r1_in;
    (* dont_touch = "true" *) wire a0, a1;
    (* dont_touch = "true" *) reg p;
    wire pre_fpa_req;

    // priority network
    wire r0 = rstn & ((~p & req0) | (p & req0 & ~req1));
    wire r1 = rstn & ((p & req1) | (~p & ~req0 & req1));

    // async latch
    assign r0_in = r0 & ~a1;
    assign r1_in = r1 & ~a0;
    muller_c_element c0 (.rstn(rstn), .A(r0_in), .B(ack_fpa), .Y(a0));
    muller_c_element c1 (.rstn(rstn), .A(r1_in), .B(ack_fpa), .Y(a1));
    assign ack0 = ~a0;
    assign ack1 = ~a1;

    // priority network (selector)
    always@(negedge ack_fpa or negedge rstn) begin
        if (!rstn)
            p <= 1'b0;
        else if (a0)
            p <= 1'b1;  // switches priority over to req1
        else if (a1)
            p <= 1'b0;  // switches priority over to req0
    end

    assign pre_fpa_req = a0 | a1;
    buffer #(.NUM_BUFFERS(NUM_BUFFERS)) fpa_buffer (.in(pre_fpa_req), .out(fpa_req));

    // data bus (origin select)
    (* dont_touch = "true" *) wire [WIDTH-1:0] data = (a1) ? data1 : data0;
    data_latch #(.WIDTH(WIDTH)) d_latch (
        .rstn(rstn),             // reset
        .req(fpa_req), .D(data), // D-Latch inputs
        .Q(data_out)             // D-Latch output
    );
```

```
60
61 endmodule
```

A.5 FPA Tree (4×1)

```
1  `timescale 1ns / 1ps
2
3  module fpa_btree #(
4      parameter N           = 4,
5      parameter WIDTH       = 2,  // log_2(N)
6      parameter NUM_BUFFERS = 0
7  ) (
8      input  wire               rstn,
9
10     input  wire [N-1:0]       req,
11     output wire [N-1:0]       ack,
12
13     output wire               root_req,      // request to readout
14     output wire [WIDTH-1:0]   data_out,      // winning channel
15     input  wire               ack_root       // readout handshake back
16 );
17
18     // interconnection variables
19     wire             req_n0, req_n1;
20     wire             ack_n0, ack_n1;
21     wire [WIDTH-1:0] data_n0, data_n1;
22
23     // Ch0/1
24     fpa_2to1 #(.WIDTH(WIDTH), .NUM_BUFFERS(NUM_BUFFERS)) FPA_01(
25         .rstn(rstn),
26         .req0 (req[0]), .req1 (req[1]),
27         .data0(2'b00),  .data1(2'b01),
28         .ack0 (ack[0]), .ack1 (ack[1]),
29         .fpa_req (req_n0),
30         .data_out(data_n0),
31         .ack_fpa (ack_n0)
32     );
33
34     // Ch2/3
35     fpa_2to1 #(.WIDTH(WIDTH), .NUM_BUFFERS(NUM_BUFFERS)) FPA_23(
36         .rstn(rstn),
37         .req0 (req[2]), .req1 (req[3]),
38         .data0(2'b10),  .data1(2'b11),
39         .ack0 (ack[2]), .ack1 (ack[3]),
40         .fpa_req (req_n1),
41         .data_out(data_n1),
42         .ack_fpa (ack_n1)
43     );
44
45     // Root
```

```
    fpa_2to1 #(.WIDTH(WIDTH), .NUM_BUFFERS(1)) FPA_RT(
        .rstn(rstn),
        .req0 (req_n0),  .req1 (req_n1),
        .data0(data_n0), .data1(data_n1),
        .ack0 (ack_n0),  .ack1 (ack_n1),
        .fpa_req (root_req),
        .data_out(data_out),
        .ack_fpa (ack_root)
    );

endmodule
```

A.6 RRA Tree (4×1)

```
`timescale 1ns / 1ps

module rra_btree #(
    parameter N           = 4,
    parameter WIDTH       = 2,  // log_2(N)
    parameter NUM_BUFFERS = 0
) (
    input  wire              rstn,

    input  wire [N-1:0]      req,
    output wire [N-1:0]      ack,

    output wire              root_req,      // request to readout
    output wire [WIDTH-1:0]  data_out,      // winning channel
    input  wire              ack_root       // readout handshake back
);

    // interconnection variables
    wire             req_n0, req_n1;
    wire             ack_n0, ack_n1;
    wire [WIDTH-1:0] data_n0, data_n1;

    // Ch0/1
    rra_2to1 #(.WIDTH(WIDTH), .NUM_BUFFERS(NUM_BUFFERS)) RRA_01(
        .rstn(rstn),
        .req0 (req[0]), .req1 (req[1]),
        .data0(2'b00),  .data1(2'b01),
        .ack0 (ack[0]), .ack1 (ack[1]),
        .fpa_req (req_n0),
        .data_out(data_n0),
        .ack_fpa (ack_n0)
    );

    // Ch2/3
    rra_2to1 #(.WIDTH(WIDTH), .NUM_BUFFERS(NUM_BUFFERS)) RRA_23(
        .rstn(rstn),
```

```
        .req0 (req[2]), .req1 (req[3]),
        .data0(2'b10),  .data1(2'b11),
        .ack0 (ack[2]), .ack1 (ack[3]),
        .fpa_req (req_n1),
        .data_out(data_n1),
        .ack_fpa (ack_n1)
    );

    // Root
    rra_2to1 #(.WIDTH(WIDTH), .NUM_BUFFERS(1)) RRA_RT(
        .rstn(rstn),
        .req0 (req_n0),  .req1 (req_n1),
        .data0(data_n0), .data1(data_n1),
        .ack0 (ack_n0),  .ack1 (ack_n1),
        .fpa_req (root_req),
        .data_out(data_out),
        .ack_fpa (ack_root)
    );

endmodule
```

A.7 Asynchronous Counter

```
`timescale 1ns/1ps

module count_async #(
    parameter N          = 4,    // number of channels
    parameter WIDTH      = 2,    // origin width
    parameter ARB_DELAY  = 6,    // ack latency
    parameter CNT_DELAY  = 6     // count latency
) (
    input  wire              resetn,

    input  wire              grant,        // channel grant arrival
    input  wire [WIDTH-1:0]  origin,       // channel origin arrival
    output wire              grant_ack,    // ack to arbiter

    output wire [32*N-1:0]   counts_async  // total counts with all channels
);

    // Splits delays passed down from top by half
    // half to origin delay, and the other half to the arbiter delay
    localparam ORG_DELAY = (ARB_DELAY >= 2) ? (ARB_DELAY / 2) : 0;

    // timers for when origin and count are ready to be received
    wire origin_clk;
    wire cnt_clk;

    // registers for origin and count
    (* keep = "true" *) reg  [WIDTH-1:0]  origin_r = {WIDTH{1'b0}};
```

```
28      (* keep = "true" *) reg  [31:0]       cnt [0:N-1];
29
30      // Arbiter Ack Delay
31      buffer #(.NUM_BUFFERS(ORG_DELAY))              origin_buffer (.in(grant),      .
        out(origin_clk));
32      buffer #(.NUM_BUFFERS(ARB_DELAY - ORG_DELAY))  arbitr_buffer (.in(origin_clk), .
        out(grant_ack));
33
34      // Async Count Delay
35      buffer #(.NUM_BUFFERS(CNT_DELAY))         cnt_buffer  (.in(grant_ack),  .out(
        cnt_clk));
36
37      // Origin capture (half less delay than grant delay)
38      always @(posedge origin_clk or negedge resetn) begin
39          if (!resetn)
40              origin_r <= {WIDTH{1'b0}};
41          else
42              origin_r <= origin;
43      end
44
45  // This allows FPA/RRA Tree and Cells to work together in one project workspace
46  generate
47  // ================================================================
48  // N = 4
49  // ================================================================
50
51      if (N == 4) begin: Trees // or No Arbiter (4x1)
52          // Async Counters
53          always @(posedge cnt_clk or negedge resetn) begin
54              if (!resetn) begin
55                  cnt[0] <= 32'd0;
56                  cnt[1] <= 32'd0;
57                  cnt[2] <= 32'd0;
58                  cnt[3] <= 32'd0;
59              end else begin
60                  if (origin_r == 2'b00)
61                      cnt[0] <= cnt[0] + 32'b1;
62                  if (origin_r == 2'b01)
63                      cnt[1] <= cnt[1] + 32'b1;
64                  if (origin_r == 2'b10)
65                      cnt[2] <= cnt[2] + 32'b1;
66                  if (origin_r == 2'b11)
67                      cnt[3] <= cnt[3] + 32'b1;
68              end
69          end
70
71          // Output bus for counters
72          assign counts_async[31:0]   = cnt[0];
73          assign counts_async[63:32]  = cnt[1];
74          assign counts_async[95:64]  = cnt[2];
75          assign counts_async[127:96] = cnt[3];
76      end
77
78  // ================================================================
```

```
// N = 2
// ================================================================

    else if (N == 2) begin: Cells // or No Arbiter (2x1)
        // Async Counters
        always @(posedge cnt_clk or negedge resetn) begin
            if (!resetn) begin
                cnt[0] <= 32'd0;
                cnt[1] <= 32'd0;
            end else begin
                if (origin_r == 1'b0)
                    cnt[0] <= cnt[0] + 32'b1;
                if (origin_r == 1'b1)
                    cnt[1] <= cnt[1] + 32'b1;
            end
        end

        // Output bus for counters
        assign counts_async[31:0]   = cnt[0];
        assign counts_async[63:32]  = cnt[1];
    end

endgenerate

endmodule
```

A.8 Synchronous Counter

```
`timescale 1ns / 1ps

module count_sync (
    input  wire        clk,           // 100MHz AXI clock domain
    input  wire        resetn,        // active low reset
    input  wire [31:0] count_async,   // raw count from async counter
    output reg  [31:0] count_sync_out  // synchronized binary count
);

// ================================================================
// ASYNCHRONOUS DOMAIN
// ================================================================

// Binary to Gray code
// -> Only 1 bit changes per increment
wire [31:0] gray_async;

genvar i;
generate
    for (i = 0; i < 31; i = i + 1) begin : b2g
        assign gray_async[i] = count_async[i] ^ count_async[i+1];
    end
```

```
endgenerate
assign gray_async[31] = count_async[31];

// ========================================================================

// ------------------------------------------------------------------------
// NO MANS LAND - the lands of inbetween
// ------------------------------------------------------------------------

// Two flip-flop synchronizers (CDC)
reg [31:0] gray_sync1, gray_sync2;

always @(posedge clk or negedge resetn) begin
    if (!resetn) begin
        gray_sync1 <= 32'd0;
        gray_sync2 <= 32'd0;
    end else begin
        gray_sync1 <= gray_async;   // First FF  - may be metastable
        gray_sync2 <= gray_sync1;   // Second FF - insurance
    end
end

// ------------------------------------------------------------------------

// ========================================================================
// SYNCHRONOUS DOMAIN
// ========================================================================

// Gray code to Binary
wire [31:0] binary_sync;

assign binary_sync[31] = gray_sync2[31];

genvar j;
generate
    for (j = 30; j >= 0; j = j - 1) begin : g2b
        assign binary_sync[j] = binary_sync[j+1] ^ gray_sync2[j];
    end
endgenerate

// Register the final output
always @(posedge clk or negedge resetn) begin
    if (!resetn)
        count_sync_out <= 32'd0;
    else
        count_sync_out <= binary_sync;
end

// ========================================================================

endmodule
```

A.9 Complete Source Code Repository

https://github.com/Kurisby/Thesis-AsyncArbiters/tree/main

Appendix B: Complete Edge-Case Counts

No Arbiter (4×1) – 30 ns Period

	Phase Pattern							
Active	`----`	`0---`	`-1--`	`01--`	`--2-`	`0-2-`	`-12-`	`012-`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**
`0-2-`	**1,000,000**	2,000,000	**1,000,000**	2,000,000	2,000,000	**1,000,000**	2,000,000	**1,000,000**
`-12-`	**1,000,000**	**1,000,000**	2,000,000	2,000,000	2,000,000	2,000,000	**1,000,000**	**1,000,000**
`0--3`	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000
`-1-3`	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000
`--23`	**1,000,000**	**1,000,000**	**1,000,000**	**1,000,000**	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**
`01-3`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**
`0-23`	**1,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**
`-123`	**1,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**
`0123`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**

	Phase Pattern							
Active	`---3`	`0--3`	`-1-3`	`01-3`	`--23`	`0-23`	`-123`	`0123`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**
`0-2-`	**1,000,000**	2,000,000	**1,000,000**	2,000,000	2,000,000	**1,000,000**	2,000,000	**1,000,000**
`-12-`	**1,000,000**	**1,000,000**	2,000,000	2,000,000	2,000,000	2,000,000	**1,000,000**	**1,000,000**
`0--3`	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**
`-1-3`	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	**1,000,000**	**1,000,000**	**1,000,000**	**1,000,000**
`012-`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**
`01-3`	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**
`0-23`	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**1,000,000**
`-123`	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**1,000,000**
`0123`	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**

Table B.1: No Arbiter (4×1) total counts in edge cases (period **30 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle.

FPA Tree (4×1) – 30 ns Period

	Phase Pattern							
Active	`----`	`0---`	`-1--`	`01--`	`--2-`	`0-2-`	`-12-`	`012-`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0123`	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000

	Phase Pattern							
Active	`---3`	`0--3`	`-1-3`	`01-3`	`--23`	`0-23`	`-123`	`0123`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0123`	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000

Table B.2: FPA Tree (4×1) total counts in edge cases (period **30 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle.

RRA Tree (4×1) – 30 ns Period

	Phase Pattern							
Active	`----`	`0---`	`-1--`	`01--`	`--2-`	`0-2-`	`-12-`	`012-`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	**2,999,996**	**2,999,966**	**2,999,999**	**2,999,999**	3,000,000	3,000,000	**2,000,000**	**2,000,000**
`0123`	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000

	Phase Pattern							
Active	`---3`	`0--3`	`-1-3`	`01-3`	`--23`	`0-23`	`-123`	`0123`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	**2,000,000**	**2,000,000**	3,000,000	3,000,000	**2,999,999**	**2,999,999**	**2,999,998**	**3,000,007**
`0123`	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000	4,000,000

Table B.3: RRA Tree (4×1) total counts in edge cases (period **30 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle.

No Arbiter (4×1) – 20 ns Period

	Phase Pattern							
Active	`----`	`0---`	`-1--`	`01--`	`--2-`	`0-2-`	`-12-`	`012-`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**
`0-2-`	**1,000,000**	2,000,000	**1,000,000**	2,000,000	2,000,000	**1,000,000**	2,000,000	**1,000,000**
`-12-`	**1,000,000**	**1,000,000**	2,000,000	2,000,000	2,000,000	2,000,000	**1,000,000**	**1,000,000**
`0--3`	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000
`-1-3`	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000
`--23`	**1,000,000**	**1,000,000**	**1,000,000**	**1,000,000**	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**
`01-3`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**
`0-23`	**1,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**
`-123`	**1,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**
`0123`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**

	Phase Pattern							
Active	`---3`	`0--3`	`-1-3`	`01-3`	`--23`	`0-23`	`-123`	`0123`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**
`0-2-`	**1,000,000**	2,000,000	**1,000,000**	2,000,000	2,000,000	**1,000,000**	2,000,000	**1,000,000**
`-12-`	**1,000,000**	**1,000,000**	2,000,000	2,000,000	2,000,000	2,000,000	**1,000,000**	**1,000,000**
`0--3`	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**	2,000,000	**1,000,000**
`-1-3`	2,000,000	2,000,000	**1,000,000**	**1,000,000**	2,000,000	2,000,000	**1,000,000**	**1,000,000**
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	**1,000,000**	**1,000,000**	**1,000,000**	**1,000,000**
`012-`	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**
`01-3`	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**
`0-23`	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**2,000,000**	**1,000,000**
`-123`	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**	**1,000,000**
`0123`	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**2,000,000**	**1,000,000**

Table B.4: No Arbiter (4×1) total counts in edge cases (period **20 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle.

FPA Tree (4×1) – 20 ns Period

	Phase Pattern							
Active	`----`	`0---`	`-1--`	`01--`	`--2-`	`0-2-`	`-12-`	`012-`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0123`	**3,165,146**	**3,155,152**	**3,159,048**	**3,165,260**	**3,165,248**	**3,157,965**	**3,158,091**	**3,165,117**

	Phase Pattern							
Active	`---3`	`0--3`	`-1-3`	`01-3`	`--23`	`0-23`	`-123`	`0123`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0123`	**3,165,194**	**3,157,689**	**3,158,330**	**3,165,247**	**3,165,284**	**3,159,401**	**3,155,240**	**3,165,320**

Table B.5: FPA Tree (4×1) total counts in edge cases (period **20 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle.

RRA Tree (4×1) – 20 ns Period

	Phase Pattern							
Active	`----`	`0---`	`-1--`	`01--`	`--2-`	`0-2-`	`-12-`	`012-`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	**2,999,989**	**2,999,988**	**2,999,999**	**2,999,999**	3,000,000	3,000,000	**2,000,000**	**2,000,000**
`0123`	**3,000,001**	**3,715,228**	**3,720,838**	**3,600,001**	**3,000,001**	**3,779,170**	**3,778,714**	**3,000,005**

	Phase Pattern							
Active	`---3`	`0--3`	`-1-3`	`01-3`	`--23`	`0-23`	`-123`	`0123`
`0---`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`-1--`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`--2-`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`---3`	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000	1,000,000
`01--`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0-2-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-12-`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`0--3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`-1-3`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`--23`	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000	2,000,000
`012-`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`01-3`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`0-23`	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000	3,000,000
`-123`	**2,000,001**	**2,000,001**	3,000,000	3,000,000	**2,999,999**	**2,999,999**	3,000,000	**2,999,994**
`0123`	**3,000,001**	**3,778,792**	**3,778,594**	**3,000,006**	**3,600,000**	**3,720,456**	**3,715,402**	**3,000,001**

Table B.6: RRA Tree (4×1) total counts in edge cases (period **20 ns**, 1M events per channel). Shaded cells have channels arriving at the same clock cycle.

Bibliography

[1] E. Aguénounon, S. Razavinejad, J.-B. Schell, M. Dolatpoor Lakeh, W. Khaddour, F. Dadouche, J.-B. Kammerer, L. Fesquet, and W. Uhring, "Design and characterization of an asynchronous fixed priority tree arbiter for SPAD array readout," *Sensors*, vol. 21, no. 12, Jun. 2021, art. no. 3949.

[2] D. Bronzi, F. Villa, S. Tisa, A. Tosi, and F. Zappa, "SPAD figures of merit for photon-counting, photon-timing, and imaging applications: A review," *IEEE Sensors Journal*, vol. 16, no. 1, pp. 3–12, 2016.

[3] S. Cova, M. Ghioni, A. Lacaita, C. Samori, and F. Zappa, "Avalanche photodiodes and quenching circuits for single-photon detection," *Applied Optics*, vol. 35, no. 12, pp. 1956–1976, 1996.

[4] Digilent, "Eclypse Z7 hardware reference manual," https://digilent.com/reference/programmable-logic/eclypse-z7/reference-manual.

[5] ——, "Eclypse Z7 Zmod AWG demo," https://digilent.com/reference/programmable-logic/eclypse-z7/demos/zmod-awg, digilent Reference.

[6] ——, "Eclypse Z7: Zynq-7000 SoC development board for modular instrumentation," https://digilent.com/shop/eclypse-z7/, product page; top-view image.

[7] Digilent, Inc., *Digilent Pmod Interface Specification*, Digilent, Inc., Pullman, WA, Oct. 2017, version 1.2.0. [Online]. Available: https://digilent.com/reference/_media/reference/pmod/digilent-pmod-interface-specification.pdf

[8] R. Donohue, "Synchronization in digital logic circuits," Lecture slides, EE183, Stanford University. https://web.stanford.edu/class/ee183/handouts/synchronization_pres.pdf.

[9] A. Forencich, "verilog-axi: Verilog AXI components for FPGA implementation," https://github.com/alexforencich/verilog-axi, 2018, gitHub repository, file `rtl/axi_register.v`.

[10] Laser Components, *Single Photon Counting Module COUNT-Series*, Laser Components Germany GmbH, 2024, datasheet. https://www.lasercomponents.com/fileadmin/user_upload/home/Datasheets/lcp/count-series.pdf.

[11] A. Mardari, Z. Jelčicová, and J. Sparsø, "Design and FPGA-implementation of asynchronous circuits using two-phase handshaking," in *Proceedings of the 2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2019, pp. 9–18.

[12] P. Papaphilippou, J. Meng, N. Gebara, and W. Luk, "Hipernetch: High-performance FPGA network switch," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 1, Nov. 2021, art. no. 3.

[13] C. Pham-Quoc and A.-V. Dinh-Duc, "Hazard-free Muller gates for implementing asynchronous circuits on Xilinx FPGA," in *Proceedings of the 2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications (DELTA)*, 2010, pp. 289–292.

[14] D. A. Santiago, "Development of an automated GPIB system for characterization of SiGe avalanche photodiodes," M.S.E. thesis, University of Nevada, Las Vegas, Las Vegas, NV, May 2023.

[15] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*. Boston, MA: Kluwer Academic Publishers, 2001.

[16] T. Turko, W. Uhring, F. Dadouche, and L. Fesquet, "An asynchronous fixed priority arbiter for high-throughput time correlated single photon counting systems," in *Proceedings of the 2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, Bordeaux, France, Dec. 2018, pp. 765–768.

[17] C.-F. E. Wu and L. M. Ni, "Asynchronous arbiter design using CMOS DCVS logic," in *1987 Symposium on VLSI Circuits, Digest of Technical Papers*, 1987, pp. 97–98.

[18] Xilinx, Inc., *UltraFast Design Methodology Guide for the Vivado Design Suite*, Xilinx, Jun. 2019, v2019.1. https://docs.amd.com/v/u/2019.1-English/ug949-vivado-design-methodology.

chrisbarr74@gmail.com

CHRIS BARR

FPGA & EMBEDDED SYSTEMS ENGINEER

SUMMARY

FPGA Engineer with experience in digital design, embedded systems integration, and hardware validation using Xilinx FPGA platforms. Proficient in Verilog and SystemVerilog development, Vivado toolchains, and low-level debugging across FPGA and embedded environments. Experienced working with communication interfaces, timing-sensitive systems, and hardware/software co-design.

TECHNICAL SKILLS

- Verilog/SystemVerilog/VHDL
- C/C++
- Python
- TCL

FPGA & EMBEDDED TOOLS

- Vivado, PetaLinux, SDK/Vitis
- WireShark
- Ubuntu/RedHat/Embedded Linux
- Git

HARDWARE PLATFORMS

- Xilinx Zynq-7000 SoC
- Altera Cyclone V

PROTOCOL & INTERFACES

- UART, SPI, I2C
- AXI, AXI-Stream
- DMA

LAB EQUIPMENT

- Oscilloscopes
- Spectrum Analyzers
- Function Generators
- Power Supplies

WORK EXPERIENCE

Engineer II

JT4 LLC, Las Vegas, NV (*October 2020 - Present*)

- Designed bare-metal FPGA application on Xilinx Zynq-7000 (ZedBoard) implementing dual time-domain switching driven by hardware switch input and internal clock, generating simulated RF RNP/ACP signals for targeted systems.
- Extended legacy Vivado hardware designs on a custom Artix-7 FPGA platform; rebuilt PetaLinux images with QSPI boot configuration and developed companion software in Vitis against the generated root filesystem.
- Deployed FPGA hardware to operational test sites and performed system-level integration, network, and instrumentation equipment, validating end-to-end functionality.
- Debugged and enhanced production Serial-to-Ethernet (vice-versa), Ethernet-to-Ethernet FPGA designs performing network routing and protocol data transformations across embedded targets.
- Built browser-based GUIs (Bootstrap, HTML/CSS) hosted directly on FPGA-served IP to interface with SDK/Vitis backend services for operator control and diagnostics; diagnosed and updated existing Nextion HMI touchscreen interfaces
- Supported large-scale test missions on-site, providing real-time coordination and analysis during active test windows.
- Automated parsing large data sets for post-data extraction, visualization, and generation of actionable graphs/reports.
- Provided technical mentorship and guidance to engineering intern on embedded systems development and debugging.
- Performed cybersecurity analysis of 2G/3G/4G mobile network traffic using Wireshark; validated fiber-optic link health and protocol integrity across distributed test systems.

EDUCATION

M.S. Electrical Engineering

University of Nevada Las Vegas | Expected: Fall 2026

- Thesis: implementation and evaluation of asynchronous arbitration on fpga for photon counting

B.S. Computer Engineering

University of Nevada Las Vegas | Graduated: Spring 2020