

# Design of a Lightweight Serial Link Generator for Test Chips

*John Wright*



Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/EECS-2017-220

<http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-220.html>

December 15, 2017

Copyright © 2017, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

---

# Design of a Lightweight Serial Link Generator for Test Chips

by John C. Wright

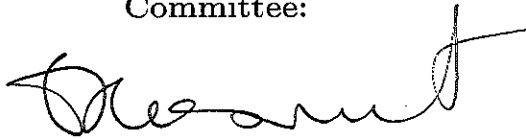
---

## Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



---

Professor Borivoje Nikolić

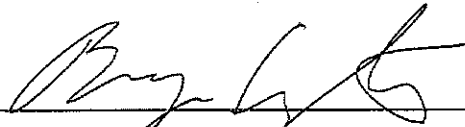
Research Advisor

12 | 11 | 17

---

Date

\*\*\*\*\*



---

Professor Vladimir Stojanović

Second Reader

12/8/17

---

Date

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Statement and Related Work . . . . .	2
1.2	Organization . . . . .	2
<b>2</b>	<b>Analog Frontend</b>	<b>3</b>
2.1	Design Specifications . . . . .	3
2.2	SPLASH2 . . . . .	4
2.2.1	Background . . . . .	4
2.2.2	ADC Interface . . . . .	4
2.2.3	TX and RX Design . . . . .	5
2.3	Hurricane 1 . . . . .	6
2.4	Hurricane 2 . . . . .	7
<b>3</b>	<b>Digital Back-end</b>	<b>10</b>
3.1	SPLASH2 . . . . .	10
3.1.1	ADC Interface . . . . .	10
3.1.2	Physical Design . . . . .	21
3.2	Hurricane 1 . . . . .	23
3.2.1	High BandWidth InterFace (HBWIF) v1.0 . . . . .	23
3.2.2	Physical Design . . . . .	29
3.3	Hurricane 2 . . . . .	30
3.3.1	High BandWidth InterFace (HBWIF) v2.0 . . . . .	30
3.3.2	Physical Design . . . . .	35
<b>4</b>	<b>Results</b>	<b>37</b>
<b>5</b>	<b>Conclusion</b>	<b>39</b>
5.1	Future Work . . . . .	39

# List of Figures

2.1	SPLASH2/Hurricane 1 frontend block diagram. . . . .	6
2.2	TX CML driver. . . . .	7
2.3	Hurricane 1 daughterboard. . . . .	8
2.4	Hurricane 2 analog IP block diagram. . . . .	9
3.1	Hittite 5831 ADC functional diagram. . . . .	11
3.2	Hittite 5831 timing diagram. . . . .	12
3.3	SPLASH2 ADC interface. . . . .	13
3.4	SPLASH2 alignment. . . . .	14
3.5	SPLASH2 XOR and word map circuit. . . . .	14
3.6	PRBS7 LFSR example. . . . .	15
3.7	Bit error rate tracker (BERT) block diagram. . . . .	16
3.8	SPLASH2 example scan bits. . . . .	18
3.9	SPLASH2 clocking diagram. . . . .	22
3.10	SPLASH2 layout. . . . .	22
3.11	HBWIF system diagram. . . . .	23
3.12	Hurricane 1 HBWIF lane micro-architecture. . . . .	24
3.13	Hurricane 1 HBWIF synchronization state machine. . . . .	26
3.14	Hurricane 1 layout. . . . .	29
3.15	Hurricane 2 TileLink switcher. . . . .	31
3.16	Hurricane 2 HBWIF lane. . . . .	32
3.17	Hurricane 2 HBWIF lane back-end (HbwifLaneBackend). . . . .	32
3.18	Hurricane 2 HBWIF lane micro-architecture (HbwifTileLinkMemSerDes). . . . .	33
3.19	Hurricane 2 DDR4 controller wrapper. . . . .	34
3.20	Hurricane 2 layout. . . . .	35
3.21	Hurricane 2 HBWIF lane layout. . . . .	36
4.1	Post-layout simulated TX eye diagram. . . . .	37
4.2	Hurricane 1 HBWIF waveforms during an FADD unit test. . . . .	38

4.3  $\log_{10}(BER)$  plot versus FPGA TX phase interpolator value (1M samples). . . 38

# Acknowledgement

I would like to thank my advisor, Bora Nikolić, for his guidance on this work and motivation to complete this report; Nandish Mehta for his receiver design; Vighnesh Iyer for his contributions to testing and FPGA design; Stevo Bailey, Ben Keller, Palmer Dabbelt, Colin Schmidt, Howie Mao, and the rest of the SPLASH2 and Hurricane teams for system-level integration, design, and verification; Andreia Cathelin and STMicroelectronics for technology access and fabrication; NASA JPL and DARPA PERFECT for funding; and my fiancée Xuan Luong for her support with everything else.

# Abstract

This report describes the design of a serial link generator using the hardware construction language Chisel and its use in the fabrication of three chips in a 28nm FDSOI process. This work includes the design of a memory interface, bit error rate tracker, analog-to-digital converter interface, and control interfaces. Silicon results demonstrate its efficacy in an Agile design methodology.



# Chapter 1

## Introduction

As with any industry, the design of integrated circuits is becoming progressively more automated, resulting in increased productivity and fewer hours spent on menial tasks. To this end, agile design methodologies [1] are becoming increasingly possible as technology platforms stabilize and IP offerings grow. Key components of agile design flows, circuit and RTL generators are useful not only for rapid prototyping, but also production-quality designs [1]. The Rocket chip generator [2] is a powerful tool for generating processor subsystems from a wide selection of tunable parameters.

Rocket chip has been used as a vehicle to build application-specific integrated circuits (ASICs) which demonstrate a variety of novel circuit techniques, including fine-grained DVFS using switched-capacitor DC-DC converters [3] and photonic interconnects [4]. Notably absent from early Rocket-chip-based ASICs has been a high bandwidth off-chip memory interface. These chips have used a slow interface known as the Host-Target InterFace (HTIF) for tethering the chip to an FPGA board, which serves as both a configuration and a memory interface [5]. This work describes a generator-based approach to high bandwidth memory interface design. By combining low-complexity serial link IP with a Rocket-chip-compatible digital back-end generator, many configurations of serial memory interfaces can be implemented rapidly, in keeping with agile design principles.

## 1.1 Problem Statement and Related Work

Memory bandwidth is a bottleneck for modern processing systems [6] [7]. Modern memory interface standards, such as DDR4 [8], are large and expensive circuits which require significant up-front non-recurring engineering (NRE) resources to complete. This NRE and/or die area can sometimes be cost prohibitive for otherwise small and inexpensive test chips.

Standards exist for chip-to-chip communication over serial links, including Interlaken [9], Aurora [10], and others. These standards will typically define a line code (a mapping of data bits to physically transmitted bits), a framing scheme, flow control mechanism, interconnect interface, and sometimes an error or redundancy check. These are frequently implemented on FPGAs using graphical tools such as Xilinx Vivado, and contain myriad configuration options selected in the Graphical User Interface (GUI). Such implementations can be classified as generators, however the reliance on GUIs and proprietary code hinder the ability to create higher-level generators with these blocks.

In this work, a Chisel[11]-based approach is presented, which allows for efficient system-level integration. A high-bandwidth interface for test chips that provides a reasonable trade-off between size/complexity/design time and performance is described. The scope of this work includes the design of the digital back-end generator and the design of the physical link. It comprises a simple serial interface used to forward memory traffic to an off-the-shelf FPGA board, leveraging the FPGA resources to emulate high bandwidth memory.

## 1.2 Organization

This work has spanned three test chips fabricated in STMicroelectronics 28nm Fully-Depleted Silicon-on-Insulator (FDSOI) technology. The three chips, from least to most recent, are SPLASH2, Hurricane 1, and Hurricane 2. Chapter 2 will discuss the three iterations of the analog frontend in chronological order. Chapter 3 will discuss the two distinctly different versions of the digital back-end (SPLASH2 vs. Hurricane 1/2).

# Chapter 2

## Analog Frontend

The analog frontend (also referred to as physical IP, frontend, and SerDes) was designed to be compatible with the voltage, current, and speed requirements of the Xilinx GTX interface [12]. The goal was to create a double-data-rate (DDR), current-mode logic (CML), source-synchronous transmitter and receiver that would work at speeds approaching 10 Gbps, but relies on the GTX interface's equalization and pre-emphasis for a good eye opening. SPLASH2 would serve as a preliminary test vehicle, with both functional and test modes available. The links would then be re-used on Hurricane chips with more complex back-ends and system integration to communicate with a Xilinx ZC706 [13] board. For the most part, the core analog components of the frontend were unchanged between chips, however cycles of learning through integration prompted changes to the digital interface of the analog IP.

### 2.1 Design Specifications

The specs for the analog frontend were chosen to comply with those in the Xilinx GTX user guide [12]. These include a maximum data rate of 10 Gbps (5 GHz), 8 lanes, 62.5 mVpp to 1.0 Vpp adjustable TX swing, 750 mV TX common-mode voltage, and 0.3 V to 0.8 V RX common-mode range.

Figure 4.1 shows an example TX eye at maximum swing with 50 mV supply ripple. These links were not designed with a specific channel model in mind, so a lumped RC load was used to simulate the receiver. Future improvements to the frontend generation will allow specific channels to be targeted (see section 5.1).

## 2.2 SPLASH2

### 2.2.1 Background

SPLASH2 is a prototype for a single-chip planetary radio-frequency spectrometer. The system comprises a serial link subsystem and a generator-based digital spectrometer, which includes a polyphase filter bank (PFB), a fast fourier transform (FFT), and an accumulator. The serial link subsystem is designed to receive raw sample data from an ADC and pack it into a larger array to be sent to the spectrometer core.

### 2.2.2 ADC Interface

The serial links communicate to a 26 GSps Analog Devices (formerly Hittite Microwave Corporation) 5831 ADC [14]. The SPLASH2 ASIC is bonded to the board and connects to the ADC or FPGA using an FPGA Mezzanine Card (FMC) connector, as shown in Figure 2.3. Details on the digital interface are in section 3.1.1 and a model of the ADC is shown in Figure 3.1. The Hittite 5831 has a negative common-mode voltage[14], so it is required to AC couple the links by inserting a series capacitor between the SPLASH2 ASIC and the ADC. The DC voltage is then set with a bias circuit on the ADC board. Because of this, it is important to maintain a “DC Balanced” signal by using the XOR pin to modulate the output. This will prevent severe inter-symbol interference (ISI) due to changes to the DC level.

### 2.2.3 TX and RX Design

The analog IP is a 10 Gbps DDR  $50\Omega$ -terminated CML transceiver (Figure 2.1), subdivided into a single TX and three RX. The transmitter comprises one differential CML driver (Figure 2.2), a digital single-ended-to-differential predriver consisting of two inverter chains, and a 2-bit to 1-bit DDR multiplexer. Each of the three receivers contains a different sampler topology (Strong Arm[15][16], Double-Tail[17], and a novel architecture). Only one of the three receivers is active at a time, the purpose being to test the performance of each separately.

There are four component circuits in the receiver datapath. First is a resistively-loaded differential amplifier (preamp) to improve sensitivity, followed by a digital sense amplifier. The sense amplifier output is fed into a latch, which converts the dynamic signal from the sense amplifier output to a static CMOS signal. Each receiver unit contains two preamps, two samplers, and two latches one each for the rising and falling edge of the clock. Finally there is a retiming circuit which aligns both the rising and falling data samples onto the rising edge of the clock, suitable for use in the digital logic back-end.

The receiver also contains a variety of configuration registers to control bias currents and clock tuning settings, which are omitted from Figure 2.1. The input to the receiver is terminated with two  $50\Omega$  resistors tied to a configurable common-mode-voltage. The common mode voltage is supplied externally, and can be bypassed to high impedance for DC-coupled applications.

The transmitter and receiver each receive reference currents (nominally 500 mA and  $60\mu\text{A}$ , respectively) provided by on-chip current mirrors. The receiver also receives an additional offset reference current of nominally  $5\mu\text{A}$ . This offset reference current is used to introduce a three-bit offset current (four thermometer coded units + sign) into the preamp differential pair to cancel any mismatch-induced offset. The transmitter reference current is supplied via 4 on-chip PMOS current mirrors, which are themselves supplied by an on-chip NMOS current mirror. The receiver currents are supplied directly from on-chip NMOS current mirrors. Each reference current mirror has an additional output current fed to a pad for testing observability.

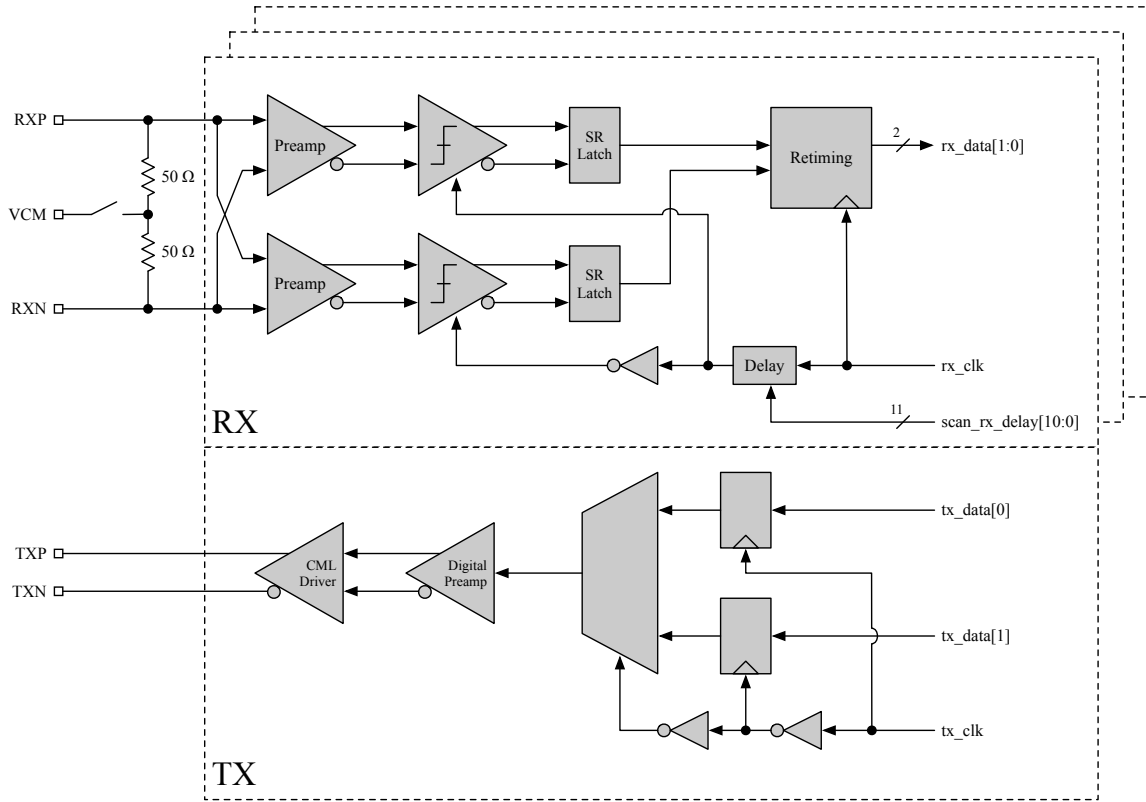


Figure 2.1: SPLASH2/Hurricane 1 frontend block diagram.

Each transceiver has a number of configuration pins which are driven by a digital back-end. In SPLASH2, a single scan chain is used to set the state of all IP on the chip. Each transceiver also has four clock pins: one for each receiver and one for the transmitter. The receiver has many configuration settings to control bias currents to the preamp, offset current, rising and falling edge clock delays, and other modes. The transmitter has far fewer- the only ones of interest being XOR (to invert the polarity of the output signals), enable (to turn the transmitters on), and swing (to adjust the output current and voltage). All of these settings are described in 3.1.1.

## 2.3 Hurricane 1

As described earlier, the purpose of these links is to interface with a ZC706 [13] FPGA board using a daughterboard (Figure 2.3). The analog blocks are unchanged from SPLASH2. Like

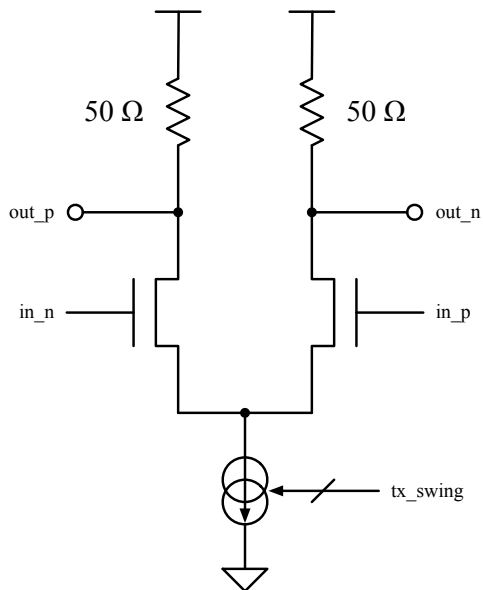


Figure 2.2: TX CML driver.

SPLASH2, Hurricane 1 also contains eight lanes. However, the current mirroring scheme is slightly modified in that there are no longer any mirrored currents going to pads for observability. The physical design is also different and is described in section 3.2.

## 2.4 Hurricane 2

Hurricane 2 contained numerous improvements to the analog block to make it more integration-friendly, reflecting the cycles of learning from SPLASH2 and Hurricane 1 (see 2.4). The IP has a single clock pin (`fast_clk`) which toggles at the line rate, and two of the redundant RX slices are removed. The serializer and deserializer have been moved into the analog IP, so that the boundary interface is the same clock domain as the back-end RTL. The clock divider is also placed inside the IP- this allows control over the skew between the clocks which drive the serializer and deserializer. This clock divider produces the clock which drives the digital back-end logic, `slow_clk`. Its data rate is a function of the digital interface's parallel data width ( $W_{data}$ ), which is 10 for Hurricane 2. The `slow_clk` frequency is determined by Equation 2.1.

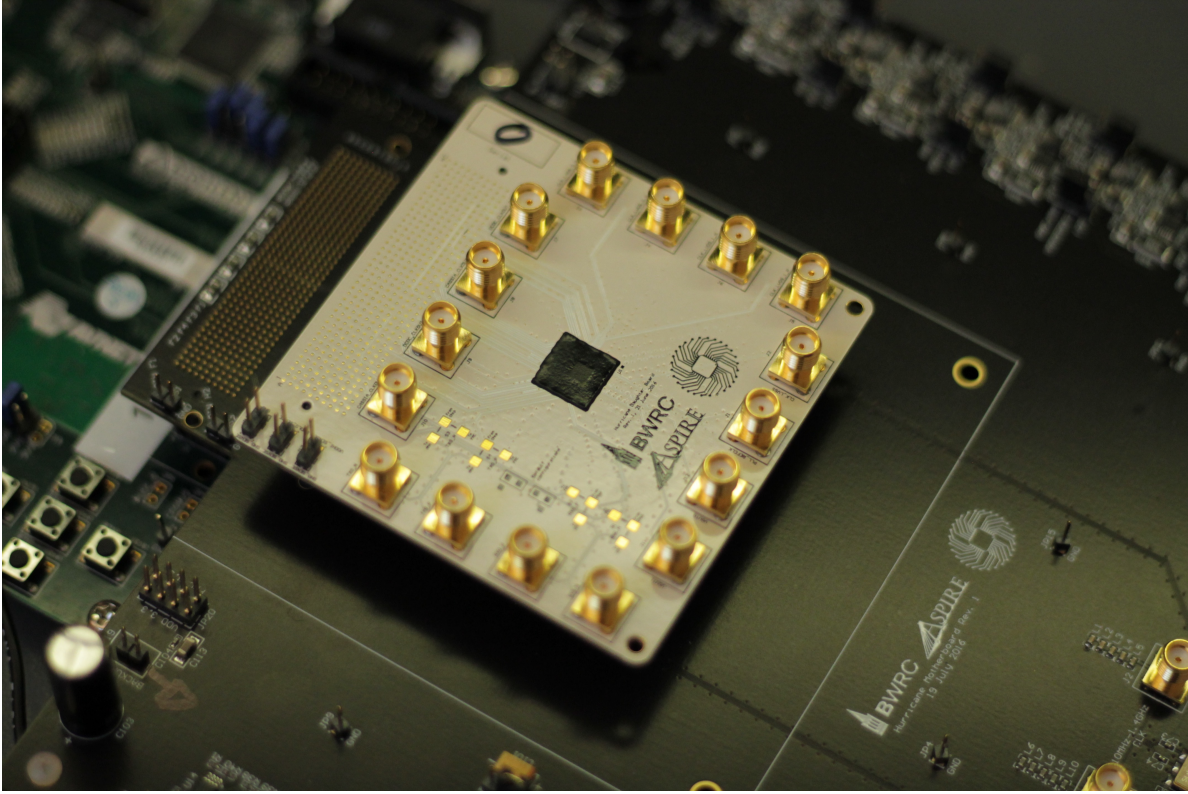


Figure 2.3: Hurricane 1 daughterboard.

$$f_{slow\_clk} = \frac{f_{fast\_clk} \cdot 2}{W_{data}} = \frac{f_{fast\_clk}}{5} \quad (2.1)$$

As a result, `slow_clk` is an output of the block, and is used as the source clock for the entire lane back-end during the lane place and route. The reset synchronizers are also placed inside the IP out of convenience, and a synchronized reset is output along with the clock. The block has an input reset signal which is driven by the primary chip-level SCR file. Figure 2.4 illustrates the components of the full Hurricane 2 analog IP. The dotted line indicates the extend of the SPLASH2/Hurricane 1 analog IP; blocks to the right of this line are new to Hurricane 2.

The three current references, TX  $I_{ref}$ , RX  $I_{ref}$ , and RX  $I_{off}$  are mirrored internally instead of provided separately. A single  $50\mu\text{A}$  reference current is required for the block, which reduces the total number of reference currents required on chip.



The lanes have been reorganized into two “banks,” each having its own chip-level reference current and common-mode voltage. This enables groupings of lanes to be independent of each other, helping with chip-level analog signal routing. As a result the chip-level current mirroring scheme needs only two four-output NMOS current mirrors.

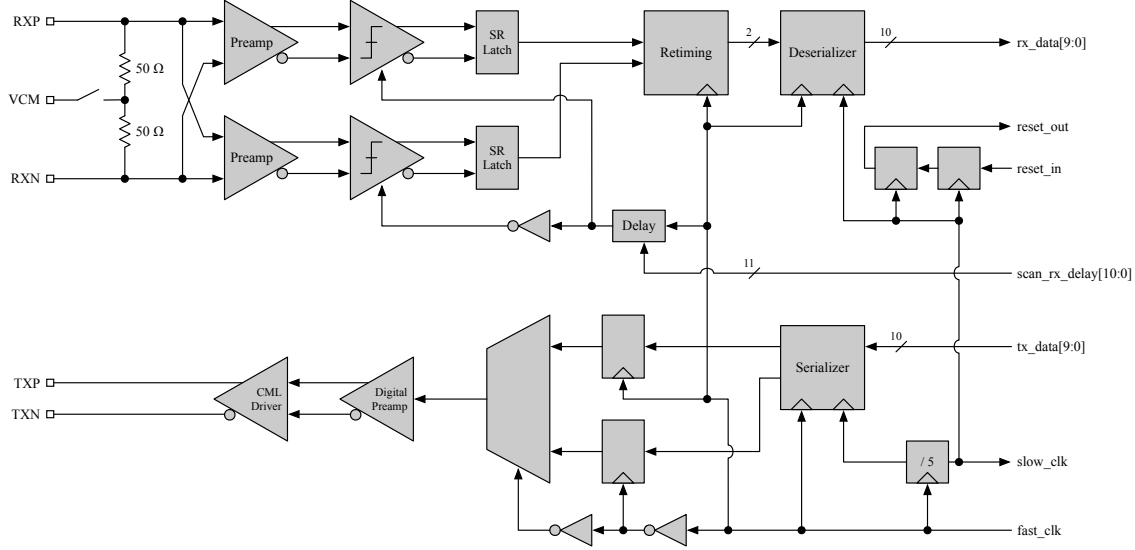


Figure 2.4: Hurricane 2 analog IP block diagram.

# Chapter 3

## Digital Back-end

This section describes the various digital back-ends created over the development of this work. All are built as generators using Chisel[11], a scala-embedded language for constructing hardware. SPLASH2 and Hurricane 1 were developed using Chisel2, and Hurricane 2 was developed using Chisel3. All extensively use a feature of Chisel called a BlackBox, which allows Analog IP and Verilog modules to be instantiated inside Chisel modules. The transceivers, serializer/deserializers (SPLASH2/Hurricane 1 only), clock dividers (SPLASH2/Hurricane 1 only), and current mirrors are all BlackBoxes. Future versions of Chisel3 will support an analog wire feature that will make integration of analog BlackBoxes like current mirrors easier.

### 3.1 SPLASH2

#### 3.1.1 ADC Interface

The ADC has two 4-bit (3-bit data + overrange) data channels which run at half the sample rate (see Figure 3.1. Each bit of the 4-bit data packet is sent in parallel over 4 differential serial links (lanes). These 4-bit data packets are demultiplexed over two physical data channels, X and Y, for a total of 8 signals (see Figure 3.2 This poses a challenge to the

back-end design, as each link is not independent from the others, so the lanes must be aligned.

**Functional Diagram**

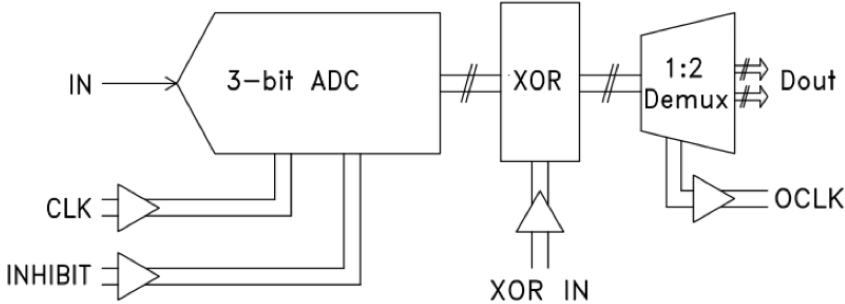


Figure 3.1: Hittite 5831 ADC functional diagram [14].

### Digital Output Format and Timing Diagram

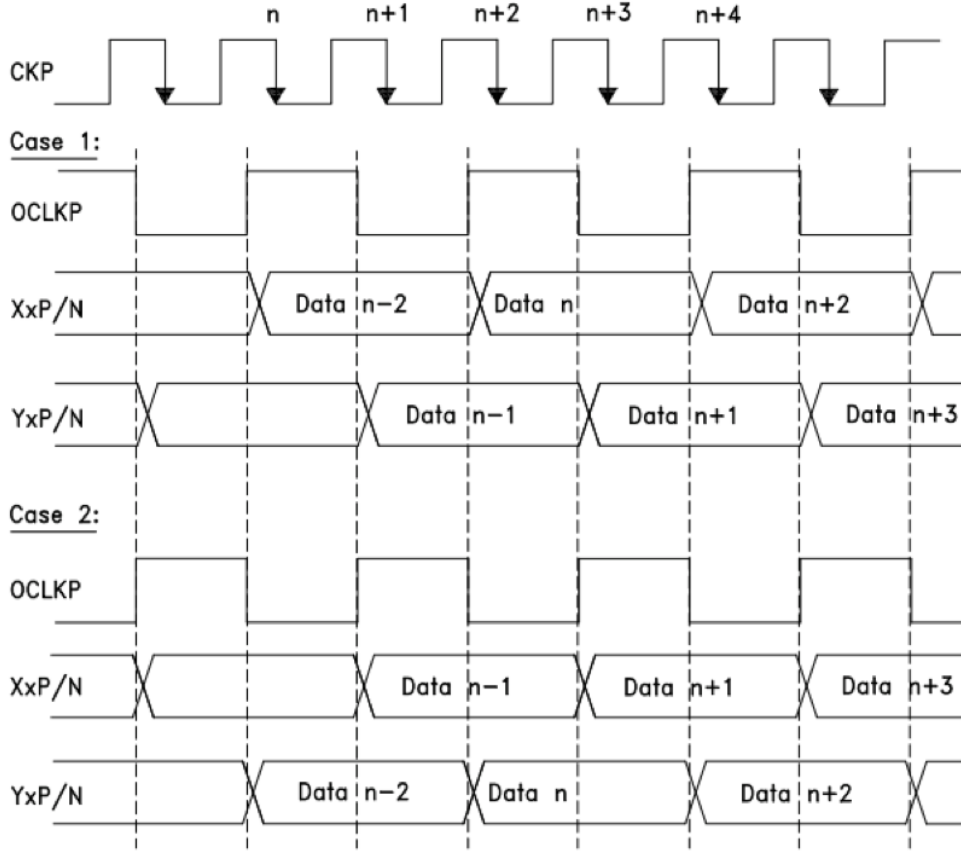


Figure 3.2: Hittite 5831 timing diagram [14].

The 5831 part provides two pins which assist in aligning the lanes: inhibit and XOR [14]. The inhibit pin will force the output data to zero, and the XOR pin will modulate the output data. Equation 3.1 shows the relationship between these pins, the output data, and the sampled data.

$$output[i] = xor \oplus (\overline{inhibit} \cdot sample[i]) \quad (3.1)$$

By sending a known XOR pattern with inhibit asserted, the back-end is able to measure the skew between links and adjust a per-lane programmable delay. Sampled data moves from the two-bit receiver interface, clocked at the fast clock frequency, to the deserializer. A divide-by-8 clock divider connects generates the slow clock from the fast clock. This

divide ratio determines the output bitwidth of the deserializer, which is  $2 \cdot R_{deser} = 16$ . The serializer input is also determined in this manner, and is also 16. The data from each lane, post-alignment, is merged into a 128-bit vector to be XOR'ed and passed to the spectrometer core. A 32-entry, 4-bit word map is included to allow use with other interfaces. This word map is implemented as a LUT, programmed via the scan chain.

Figure 3.4 illustrates the hardware that implements this algorithm. Note that the XOR signal is also used after alignment to assist in DC balancing the channels. The importance of DC balance is outlined in section 2.2.2. Figure 3.3 outlines the board-level SPLASH2 system.

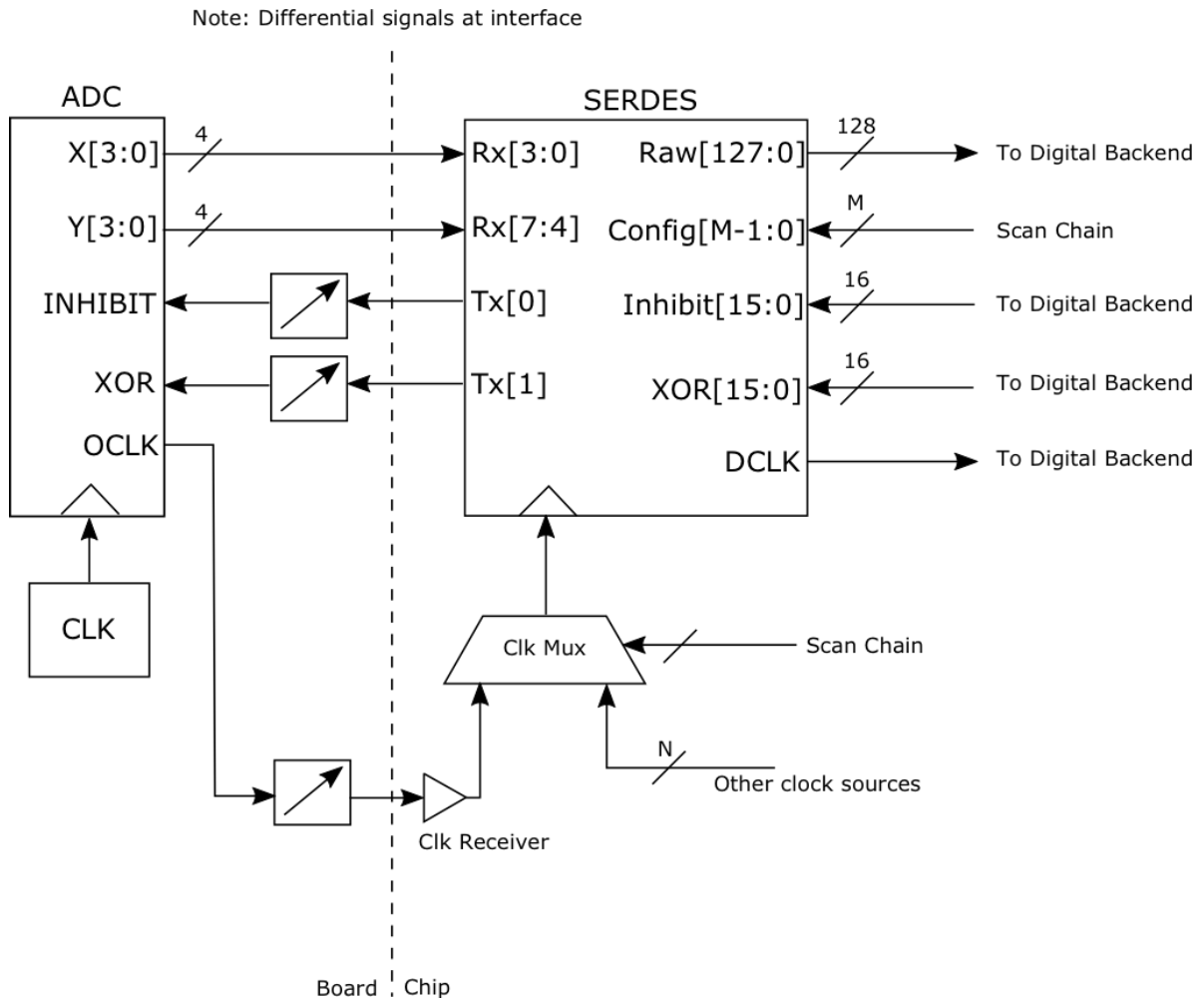


Figure 3.3: SPLASH2 ADC interface.

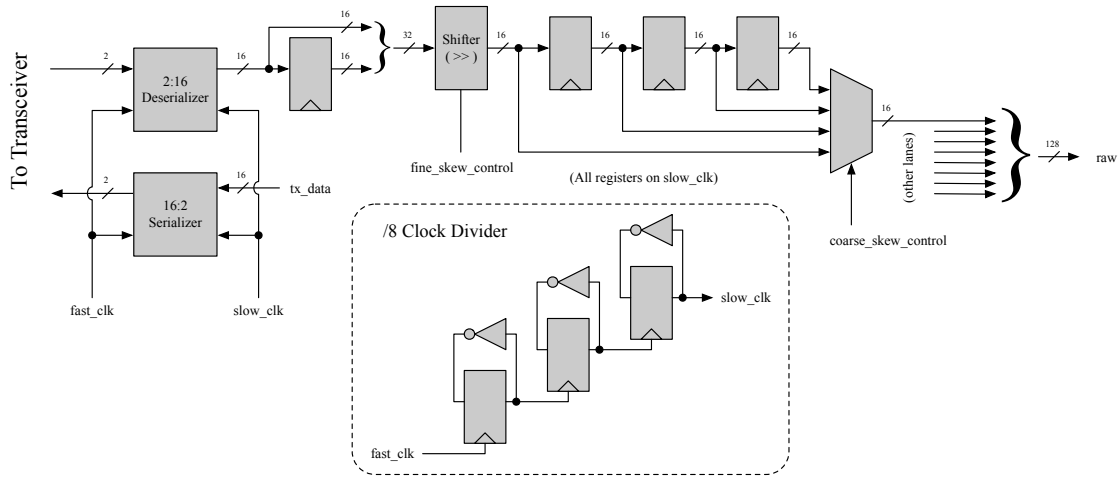


Figure 3.4: SPLASH2 alignment.

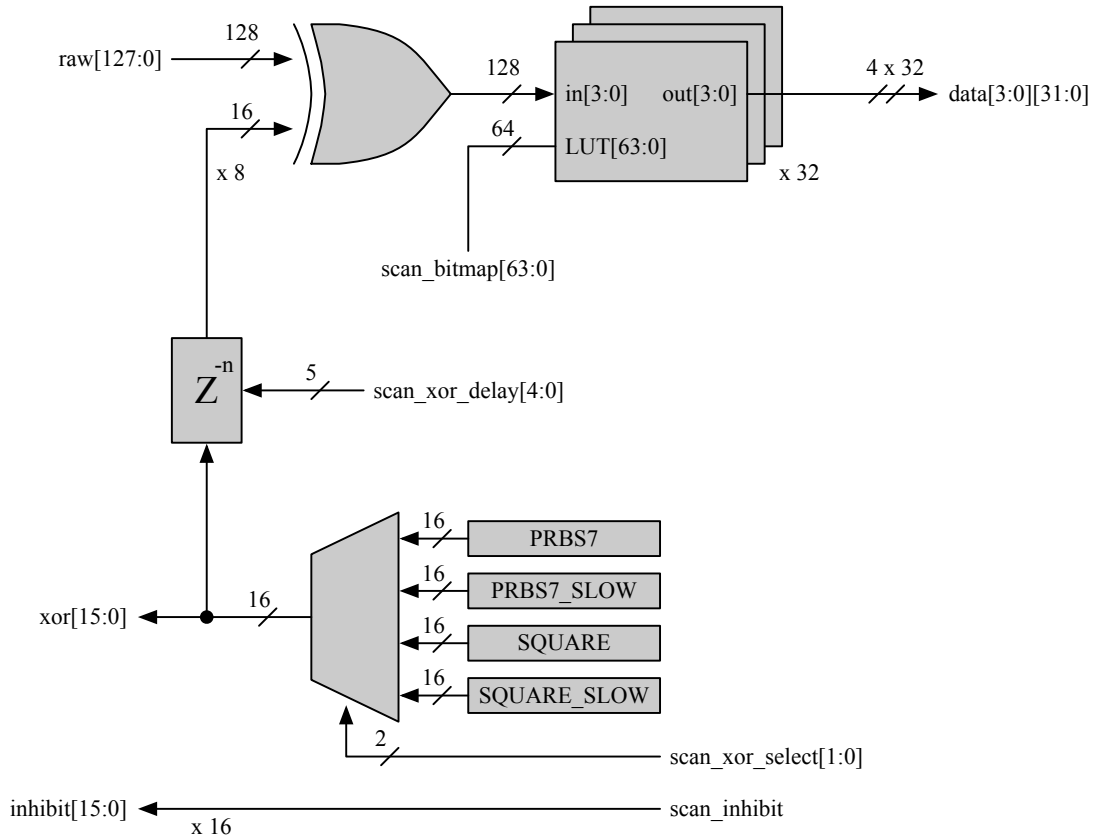


Figure 3.5: SPLASH2 XOR and word map circuit.

## Bit Error Rate Tracker (BERT)

SPLASH2 includes a single Bit Error Rate Tracker (BERT) circuit [18]. This is a test circuit which is used in place of the ADC back-end. The BERT provides three Pseudo-Random Bit Sequence (PRBS) stimuli (PRBS7, PRBS15, and PRBS31 standards). When connected to an FPGA producing an identical sequence or in loopback mode (TX jumpered to RX), the BERT circuit is put into seed mode, which uses the incoming data to set the state of a linear feedback shift register (LFSR). An example of a PRBS7 LFSR is shown in Figure 3.6

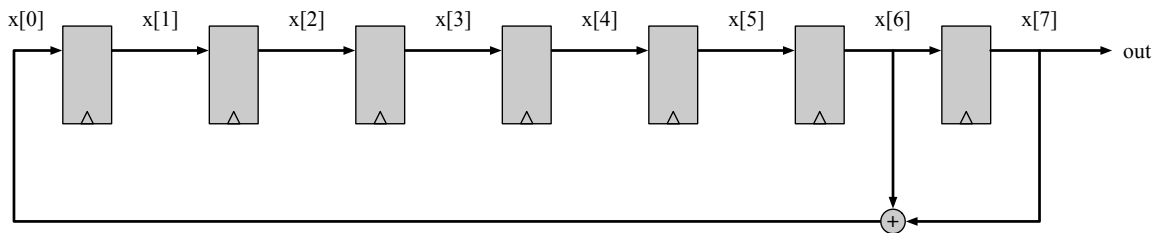


Figure 3.6: PRBS7 LFSR example.

Once the LFSR state is seeded with at least  $N$  samples ( $N$  being the order of the LFSR), the state of the transmitting LFSR is known, and future bit sequences can be calculated. Under the assumption that the seeded value is correct (which is likely true for low bit error rates), the subsequent data can be XORed with the expected data to obtain error bits, which are then accumulated over a long period of time. For PRBS7 and PRBS15, only one deserialized cycle is required for seeding since the deserialized data width (16) is larger than the LFSR order.

The implementation of BERT used on SPLASH2 offers a number of features migrated from previous work [18]. Section 3.1.1 includes every control pin and its function.

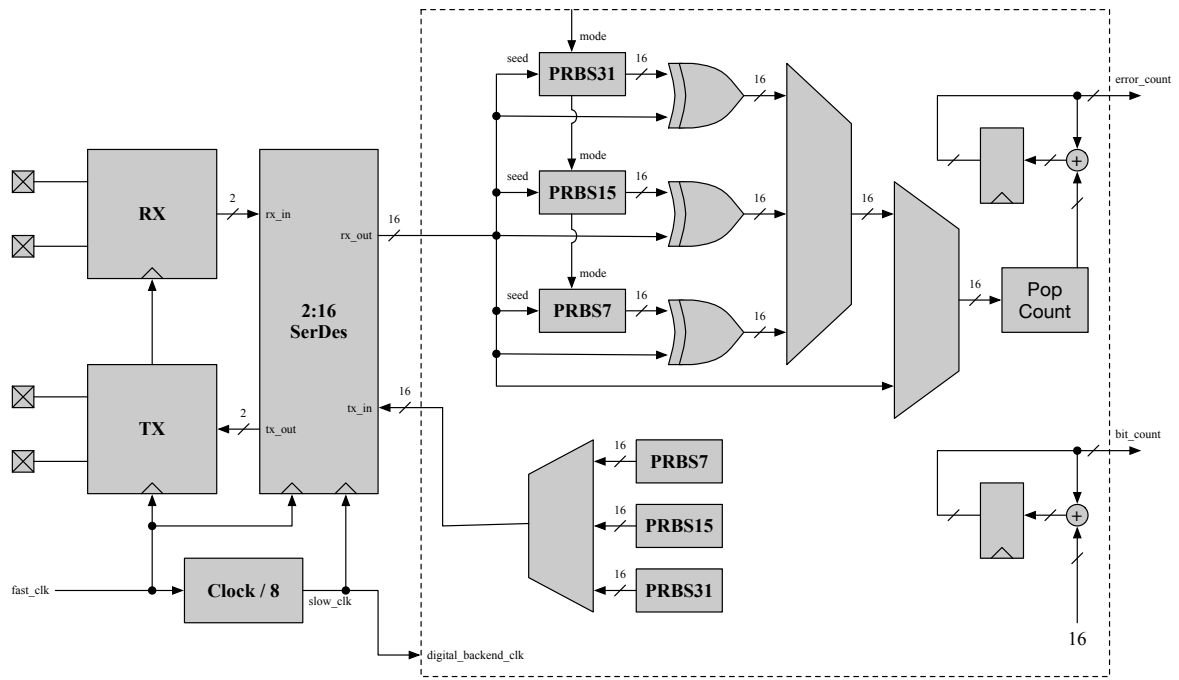


Figure 3.7: Bit error rate tracker (BERT) block diagram.

Listing 1 shows the SPLASH2 transceiver interface written in Chisel.



```

1 package hurricane_hbwif
2
3 import Chisel._
4 import ChiselError._
5
6 class TransceiverRxConfig extends Bundle {
7   // Global config bits
8   val cvd      = Bits(INPUT, width = 9)
9   val delay    = Bits(INPUT, width = 3)
10  // Slice 1 config bits
11  val ioff1    = Bits(INPUT, width = 8)
12  val ioffen1  = Bits(INPUT, width = 3)
13  val irefen1  = Bits(INPUT, width = 3)
14  val it1      = Bits(INPUT, width = 10)
15  val sw1      = Bits(INPUT, width = 4)
16  val clken1   = Bool(INPUT)
17  // Slice 2 config bits
18  val ioff2    = Bits(INPUT, width = 8)
19  val ioffen2  = Bits(INPUT, width = 3)
20  val irefen2  = Bits(INPUT, width = 3)
21  val it2      = Bits(INPUT, width = 10)
22  val sw2      = Bits(INPUT, width = 4)
23  val clken2   = Bool(INPUT)
24  // Slice 3 config bits
25  val ioff3    = Bits(INPUT, width = 8)
26  val ioffen3  = Bits(INPUT, width = 3)
27  val irefen3  = Bits(INPUT, width = 3)
28  val it3      = Bits(INPUT, width = 10)
29  val sw3      = Bits(INPUT, width = 4)
30  val rx3del   = Bits(INPUT, width = 8)
31  val clken3   = Bool(INPUT)
32  // VCM on/off switch
33  val vcmsw    = Bool(INPUT)
34 }
35
36 class TransceiverTxConfig extends Bundle {
37   val en       = Bool(INPUT)
38   val xor      = Bool(INPUT)
39   val swing    = Bits(INPUT, width = 4)
40 }
41
42 class TransceiverIO(num_clocks: Int = 4) extends Bundle {
43
44   // high speed clock input(s)
45   val clks = Vec.fill(num_clocks) { Bool(INPUT) }
46
47   // rx pad inputs
48   val rx_inp = Bool(INPUT)
49   val rx_inn = Bool(INPUT)
50
51   // tx pad inputs
52   val tx_outp = Bool(OUTPUT)
53   val tx_outn = Bool(OUTPUT)
54
55   // rx internal outputs
56   val rx_out1 = Bits(OUTPUT, width = 2)
57   val rx_out2 = Bits(OUTPUT, width = 2)
58   val rx_out3 = Bits(OUTPUT, width = 2)
59
60   // tx internal inputs
61   val tx_in = Bits(INPUT, width = 2)
62
63   // config stuff
64   val rx_config = new TransceiverRxConfig
65   val tx_config = new TransceiverTxConfig
66
67   // analog stuff
68   val rx_vcm = Bool(INPUT)
69   val tx_iref = Bool(INPUT)
70   val rx_iref = Bool(INPUT)
71   val rx_ioff = Bool(INPUT)
72 }
73
74 class Transceiver(num_clocks: Int = 4) extends BlackBox {
75   val io = new TransceiverIO(num_clocks)
76
77   moduleName = "hurricane_hbwif_top"
78 }

```

Listing 1: SPLASH2 transceiver Chisel2 BlackBox code.

## Scan Chain

The SPLASH2 scan chain is automatically generated using custom Chisel objects. A Chisel object was written to allow Chisel nets to be assigned as scan nets, and a second chisel object generates the scan registers and stitches them together. Chisel also outputs an address map of these scan registers in a variety of formats (including  $\LaTeX$ !). Listings 2 and 3 shows the Chisel code used to generate scan registers. Figure 3.1.1 shows sample scan bits for a single lane.

Name	Description	MSB	LSB	R/W
s_0_cvd	RX Clock Vernier Delay	756	764	W
s_0_delay	RX Clock Delay	765	767	W
s_0_ioff1	RX Offset Current Setting (Slice 1)	768	775	W
s_0_ioffen1	RX Offset Current Enable (Slice 1)	776	778	W
s_0_irefen1	RX Reference Current Enable (Slice 1)	779	781	W
s_0_it1	RX Tail Current Setting (Slice 1)	782	791	W
s_0_sw1	RX Current Mirror Control (Slice 1)	792	795	W
s_0_clken1	RX Clock Enable (Slice 1)	796	796	W
s_0_ioff2	RX Offset Current Setting (Slice 2)	797	804	W
s_0_ioffen2	RX Offset Current Enable (Slice 2)	805	807	W
s_0_irefen2	RX Reference Current Enable (Slice 2)	808	810	W
s_0_it2	RX Tail Current Setting (Slice 2)	811	820	W
s_0_sw2	RX Current Mirror Control (Slice 2)	821	824	W
s_0_clken2	RX Clock Enable (Slice 2)	825	825	W
s_0_ioff3	RX Offset Current Setting (Slice 3)	826	833	W
s_0_ioffen3	RX Offset Current Enable (Slice 3)	834	836	W
s_0_irefen3	RX Reference Current Enable (Slice 3)	837	839	W
s_0_it3	RX Tail Current Setting (Slice 3)	840	849	W
s_0_sw3	RX Current Mirror Control (Slice 3)	850	853	W
s_0_rx3del	RX Delay (Slice 3)	854	861	W
s_0_clken3	RX Clock Enable (Slice 3)	862	862	W
s_0_vcmsw	Common-Mode Voltage Enable	863	863	W

Figure 3.8: SPLASH2 example scan bits.

```

1 object ScanChain {
2
3   val write_regs = ListBuffer[ScanReg]()
4   val read_regs = ListBuffer[ScanReg]()
5   def regs: ListBuffer[ScanReg] = { write_regs ++ read_regs }
6   val scan_update = Bool()
7   val scan_clk = Bool()
8   val scan_in = Bool()
9   val names = HashMap[String Int]()
10
11   val read_scan_in = Bool()
12
13   def scan_out: Bool = { regs.last.io.scan_out }
14   def data_bits: Int = { regs.foldLeft(0) { (x,y) => (if(y.dummy) 0 else y.width) + x } }
15   def read_bits: Int = { read_regs.foldLeft(0) { (x,y) => x + y.width } }
16
17   def addReg(sr: ScanReg, name: String): Bits = {
18     if (sr.read_only) {
19       if (read_regs.isEmpty) {
20         sr.io.scan_in := read_scan_in
21       } else {
22         sr.io.scan_in := read_regs.last.io.scan_out
23       }
24       read_regs.append(sr)
25     } else {
26       if (write_regs.isEmpty) {
27         sr.io.scan_in := scan_in
28       } else {
29         sr.io.scan_in := write_regs.last.io.scan_out
30       }
31       write_regs.append(sr)
32     }
33   }
34   sr.io.scan_update := scan_update
35   sr.io.scan_clk := scan_clk
36
37   // check the name list
38   if (names.contains(name)) {
39     names(name) += 1
40     sr.net_name = "s_" + name + "_" + names(name).toString
41     sr.setName("ScanReg_" + name + "_" + names(name).toString)
42   } else {
43     names(name) = 1
44     sr.net_name = "s_" + name
45     sr.setName("ScanReg_" + name)
46   }
47   sr.io.data
48 }
49
50 def finish = { read_scan_in := write_regs.last.io.scan_out }
51
52 def config: ListBuffer[(String Int Int Boolean)] = {
53   var idx = 0
54   regs.map { x =>
55     idx += x.width
56     if (x.msb_to_lsb) {
57       (x.net_name, idx - 1, idx - x.width, x.read_only)
58     } else {
59       (x.net_name, idx - x.width, idx - 1, x.read_only)
60     }
61   }
62 }
63
64 def width: Int = { regs.foldLeft(0) { _ + _.width } }
65
66 // Note: removed output methods for brevity
67 }

```

Listing 2: SPLASH2 scan generator Chisel2 code.

```

1 class ScanReg(
2   val width: Int = 1,
3   //val init: Int = 0,
4   val read_only: Boolean = false,
5   val mod_type: String = null,
6   var net_name: String = "unknown",
7   val msb_to_lsb: Boolean = true,
8   val dummy: Boolean = true
9 ) extends Module {
10   val io = new Bundle {
11     val scan_clk = Bool(INPUT)
12     val scan_in = Bool(INPUT)
13     val scan_update = Bool(INPUT)
14     val scan_out = Bool(OUTPUT)
15     val data = Bits(width = width)
16   }
17
18   def setName(n: String) { name = n; named = true } // hack, this is in the current chisel version
19
20   //val slices = List.fill(width) { Module(new ScanRegSlice(init = init, read_only = read_only, mod_type = mod_type)) }
21   val slices = List.fill(width) { Module(new ScanRegSlice(read_only = read_only, mod_type = mod_type)) }
22   val scan_wires = List.fill(width+1) { UInt(width = 1) }
23
24   slices.map { _ .io.scan_clk := io.scan_clk }
25   slices.map { _ .io.scan_update := io.scan_update }
26
27   scan_wires(0) := io.scan_in
28   slices.zipWithIndex.map { x => scan_wires(x._2+1) := x._1.io.scan_out }
29   slices.zipWithIndex.map { x => x._1.io.scan_in := scan_wires(x._2) }
30
31   io.scan_out := scan_wires(width)
32   if (read_only) {
33     io.data.dir = INPUT
34     if (msb_to_lsb) {
35       slices.zipWithIndex.map { x => x._1.io.data := io.data(width-x._2-1) }
36     } else {
37       slices.zipWithIndex.map { x => x._1.io.data := io.data(x._2) }
38     }
39   } else {
40     io.data.dir = OUTPUT
41     if (msb_to_lsb) {
42       io.data := slices.slice(1,width).map { _ .io.data }.foldLeft(slices(0).io.data) { Cat(_,_) }
43     } else {
44       io.data := slices.slice(0,width-1).reverse.map { _ .io.data }.foldLeft(slices(width-1).io.data) { Cat(_,_) }
45     }
46   }
47 }

```

Listing 3: SPLASH2 scan register Chisel2 code.

### 3.1.2 Physical Design

SPLASH2 was assembled using a Synopsys tool flow. As described in section 3.1.1, the data from each link communicates to a common digital back-end, which itself communicates synchronously to the spectrometer core. A consequence of this design is that the entire chip must be synchronous. SPLASH2 has two clocks: a fast clock which toggles at the link data rate, and a slow clock which is divided by the (de)serialization factor 8. The slow clock is distributed to the digital back-end and spectrometer core, and the fast clock is distributed to the analog IPs. Figure 3.9 shows the clocking of this chip. Because of the large area of this chip and the synchronization requirements, timing targets were not met because of issues at clock crossing boundaries. This issue is fixed in future chips (see the Hurricane sections) by using a different clocking scheme.

The serializer/deserializer (SerDes) block is synthesized, placed and routed along with the rest of the RTL. While there were actually few issues with timing in this block, Hurricane 2 shows that it is easier to include the serializer and deserializer as custom logic within the transceiver macro itself. This also has the added benefit of only requiring a single sink per lane for the fast clock.

Figure 3.10 shows a layout of the die, which is 2.034 mm x 1.621 mm. The transceivers are placed around the outside of the chip, marked by magenta rectangles. The input clock receivers are marked by cyan rectangles, and the PLL is marked by a yellow rectangle.

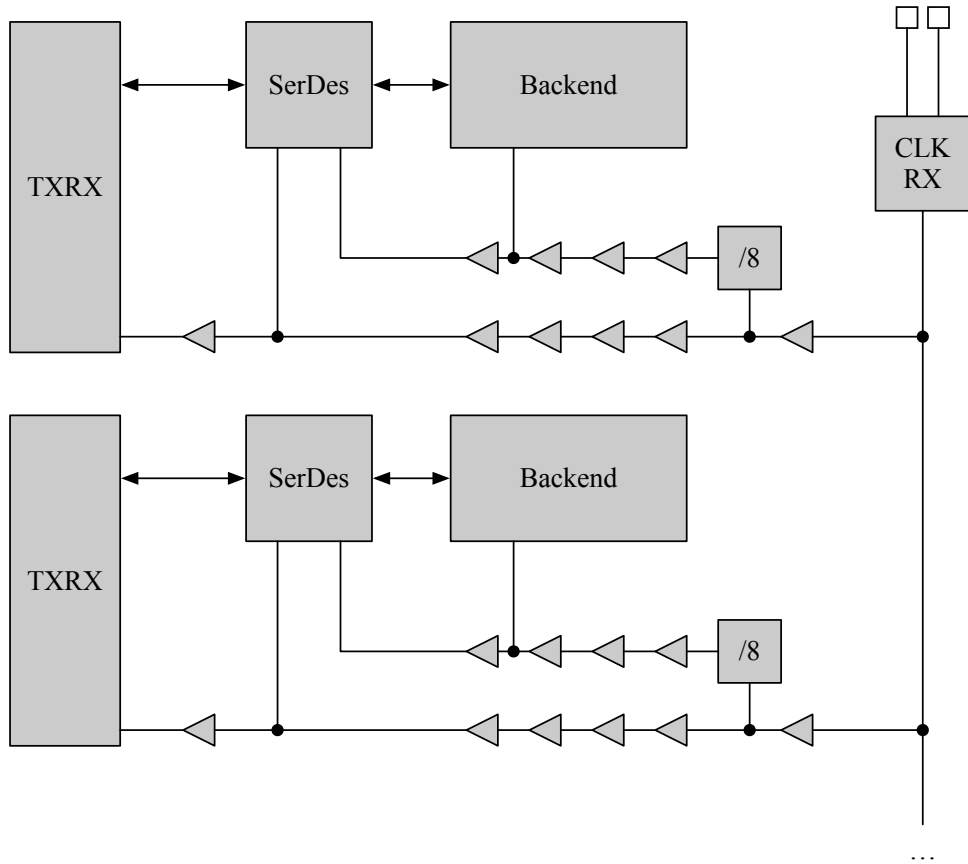


Figure 3.9: SPLASH2 clocking diagram.

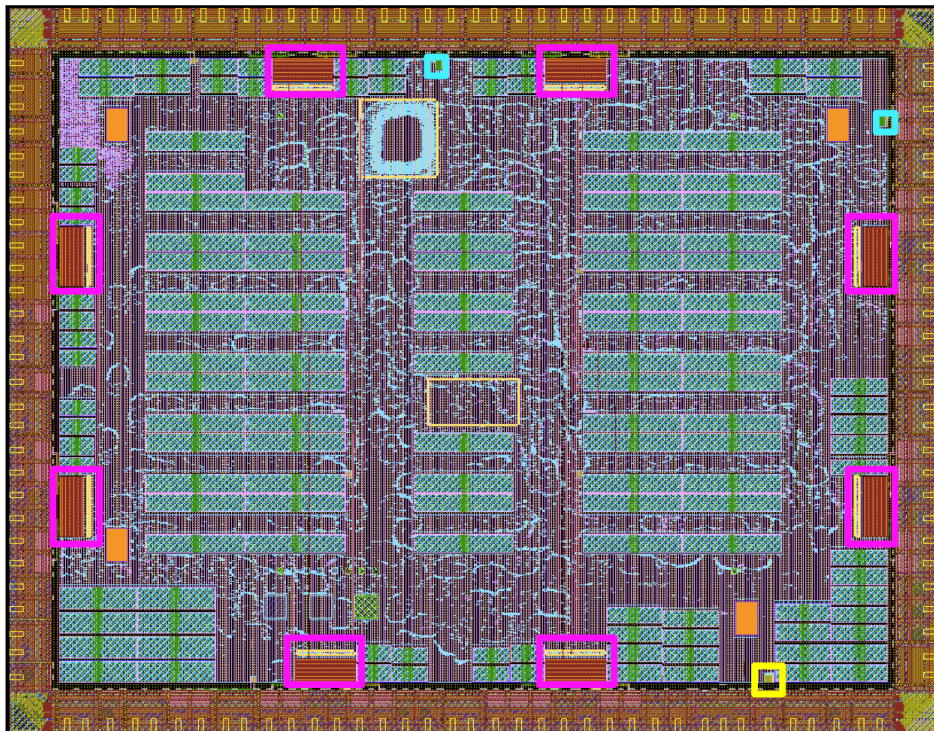


Figure 3.10: SPLASH2 layout.

## 3.2 Hurricane 1

### 3.2.1 High BandWidth InterFace (HBWIF) v1.0

Hurricane HBWIF is a tunneled NASTI-over-Serial memory interface using high-speed serial links. NASTI (Not A Standard Interconnect) is an on-chip memory interconnect protocol. HBWIF communicates to an (off-chip) FPGA via 10 Gbps DDR 50 $\Omega$ -terminated CML drivers and receivers. HBWIF has multiple memory ports which connect to the outer memory system using the NASTI interface. An example block diagram for the HBWIF use case is included in Figure 3.11.

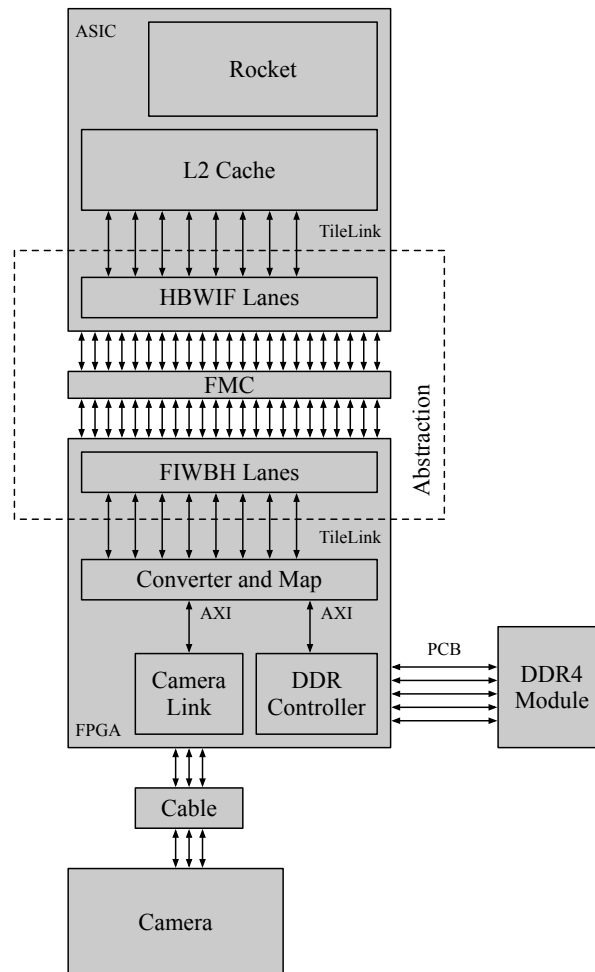


Figure 3.11: HBWIF system diagram.

There are 8 total lanes within HBWIF, each comprising a transceiver, SerDes, 8b/10b encoder, synchronization engine, and buffer (Figure 3.12). Each lane contains a number of control registers, which are configured using a Simple Memory Interface (SMI) port, which is accessed by the processor as Memory-Mapped IO (MMIO). There are two possible memory configurations for HBWIF, 1-channel and 8-channel. In the 1-channel configuration, only lane 0 is used.

HBWIF can also be bypassed (the default setting), in which case the backup memory port is used to access main memory. The backup memory port uses the HTIF as a physical layer. When HBWIF is bypassed, the lanes are in test mode, and control and status registers are still readable and writeable. During test mode, the lanes send and receive a PRBS, auto-locking to the PRBS on the receive end. Bit errors are tracked in a per-lane Bit Error Rate Tracker (BERT) module, which is accessed via the SMI interface. There is no real-time bit error tracking during memory interface mode.

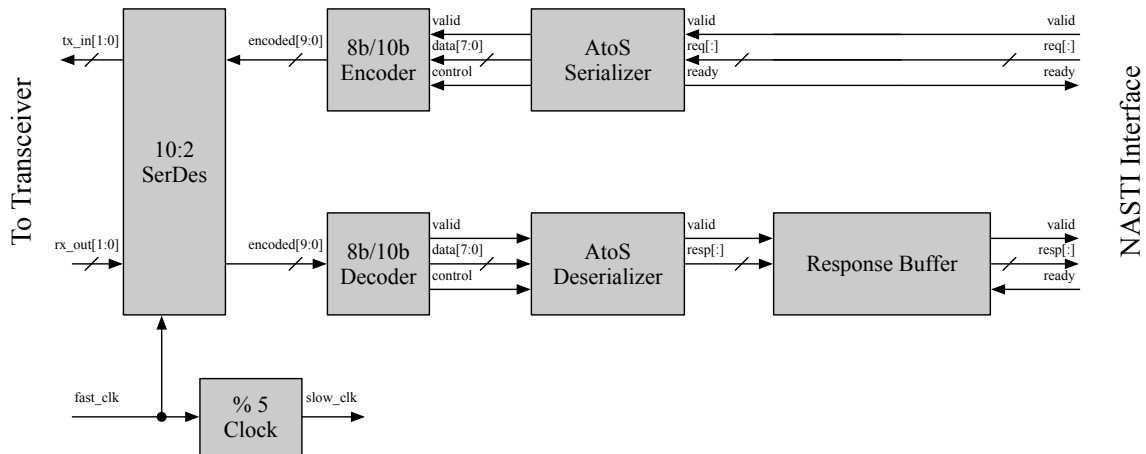


Figure 3.12: Hurricane 1 HBWIF lane micro-architecture.

### Bit Error Rate Tracker (BERT)

A bit error rate tracker (BERT) engine is included per lane. This circuit is unchanged from SPLASH2 except that the bitwidth is changed from 16 bits to 10 bits. A multiplexer is used to choose TX deserialized data from the BERT back-end or the HBWIF back-end. Both blocks receive the RX deserialized data. See 3.1.1 for more information.



## Buffer

Hurricane 1 processes NASTI memory requests as soon as they are received. A counter keeps track of the number of in-flight requests. It is incremented whenever a new request is issued and decremented whenever a response is read by the memory system. When the memory system is not ready to receive a response (i.e. `response_ready` is low), the responses are stored in a response buffer of parameterizable depth. If the total number of in-flight memory requests reaches the buffer depth, the `request_ready` signal will deassert, preventing new requests from firing until responses are dequeued from the buffer.

## Synchronization

A simple synchronization state machine is included to ensure that the FPGA is brought up and ready to receive memory traffic (Figure 3.13). The procedure is as follows:

- **Reset:** The lane waits in reset until it is put into memory mode with a write of 1 to the `mem_en` register.
- **Sync:** The lane will emit a synchronization symbol over the lane, which is encoded as an 8b/10b control word (see section 3.2.1).
- **Ack:** The lane will wait in the ack state until it receives an acknowledgment symbol from the FPGA, which is also encoded as an 8b/10b control word.
- **Idle:** Once the acknowledgement has been received, the lane is ready in the idle state. It will transition into the busy state when a valid memory request is received. During this state, `mem_req_ready` is set to 1, which allows a valid memory request to be received.
- **Busy:** After a valid memory request is received, the constituent signals of the NASTI request are converted into a single wide vector (the AtoS block in Figure 3.12). This wide vector is serialized into 8-bit bytes and sent sequentially over the TX channel. In this state `mem_req_ready` is set to 0 to disallow incoming memory requests.
- **Last:** During the final state, the last 8-bit byte is sent. This state is functionally similar to the busy state, but `mem_req_ready` is set to 1 to allow a new request to begin.

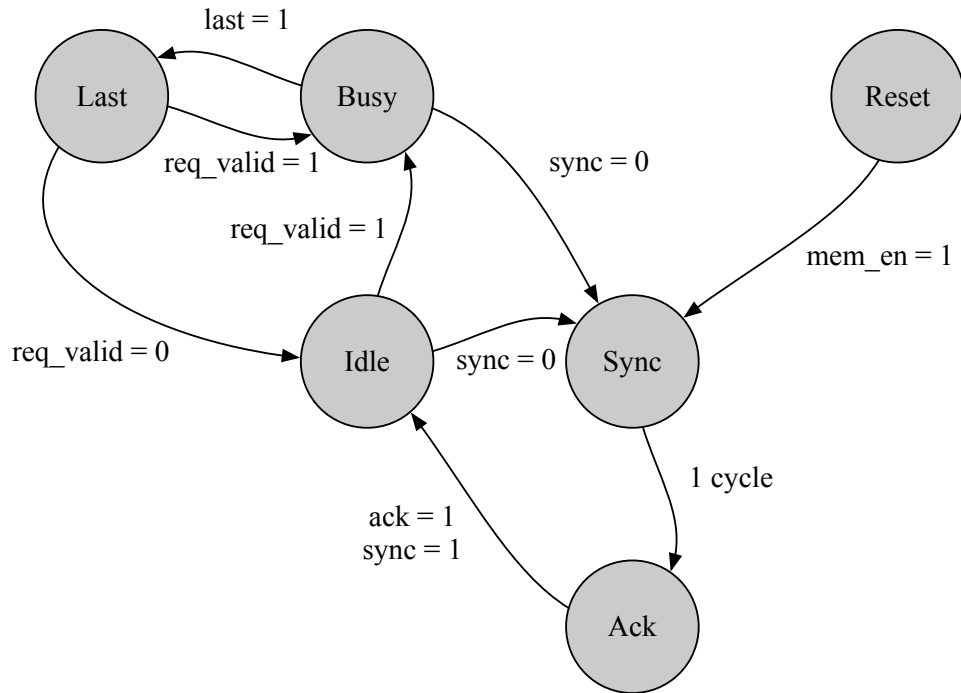


Figure 3.13: Hurricane 1 HBWIF synchronization state machine.

### 8b/10b Encoder/Decoder

HBWIF uses 8b/10b [19] as a line code to provide DC balance and word alignment. The 8b/10b code is implemented in Chisel using a `Seq` of bit patterns to create a mapping. A single register holds the running disparity (RD) state for the transmit side, and is used to select the proper encoded word to output over the TX channel of the two possible codes. In this implementation, the receive side ignores the current running disparity and simply decodes whatever 10-bit word was received. This is allowed as the 8b/10b decoding is surjective. The decoder must receive at least one comma symbol, which contains a series of five sequential 1s or five sequential 0s, to align. The `valid` signal of the decoded output will remain low until a valid comma sequence is detected. Commas between HBWIF packets are otherwise ignored.

HBWIF uses only a select few control symbols for synchronization and commas. `K.28.5` is used as the comma sequence, although any other comma besides `K.28.7` is detectable and valid for alignment. `K.28.7` is disallowed in the HBWIF implementation as it complicates

comma detection. K.28.0 is used as the SYNC symbol, and K.28.4 is used as the ACK symbol.

## **Serializer/Deserializer**

The Hurricane 1 serializer/deserializer is implemented in Verilog RTL and is placed-and-routed with other parts of the outer memory system of the chip. This block is wrapped in a Chisel BlackBox to be used within the Chisel domain. A 3-bit counter keeps track of the fast clock edge index within a slow clock cycle (which is one-fifth the fast clock rate). The slow-clock-synchronized reset is used to reset the counter to zero, which then rolls-over to zero every five cycles. This RTL is shown in Listing 4.

On every 0<sup>th</sup> fast clock cycle, the serializer latches a 10-bit word from the slow clock domain into a register clocked by the fast clock. On every other fast clock cycle, this register is shifted right by two bit positions. The serialized data is selected from the bottom two bit positions each cycle- this data feeds directly into the analog IP.

The deserializer contains a shift register that latches two bits at a time from the RX output of the analog IP. Every fast clock cycle this is shifted to the right by two bit positions, until ten total bits are captured (five cycles). A ten-bit shadow register is clocked by the slow clock and fed by the parallel output of this shift register. This register will prevent the data from changing during a slow clock cycle, which would lead to setup and hold violations in the digital back-end.

```

1  always @(posedge fast_clk) begin
2      if (io_reset) begin
3          io_rx_out <= 0;
4          count <= 0;
5          end else begin
6
7          // rx shift register
8          for(i = 1; i < NDIVBY; i = i + 1) begin
9              rx_buffer[i*2-2] <= rx_buffer[i*2];
10             rx_buffer[i*2-1] <= rx_buffer[i*2+1];
11         end
12         rx_buffer[2*NDIVBY-1] <= rx_in_d[1];
13         rx_buffer[2*NDIVBY-2] <= rx_in_d[0];
14
15         if (count == (NDIVBY-1)) begin
16             // This will hold the outputs for an entire slow clock cycle
17             io_rx_out <= rx_buffer;
18             tx_buffer <= io_tx_in;
19             count <= 0;
20         end else begin
21             count <= count + 1;
22
23             // tx shift register
24             for(i = 1; i < NDIVBY; i = i + 1) begin
25                 tx_buffer[i*2-2] <= tx_buffer[i*2];
26                 tx_buffer[i*2-1] <= tx_buffer[i*2+1];
27             end
28             tx_buffer[2*NDIVBY-1] <= 1'b0;
29             tx_buffer[2*NDIVBY-2] <= 1'b0;
30         end
31     end
32 end

```

Listing 4: Hurricane 1 serializer/deserializer RTL.

### 3.2.2 Physical Design

Figure 3.14 shows the GDS of Hurricane 1, which highlights the two Rocket cores and the L2 cache in white. The eight magenta rectangles identify the locations of the serial link analog IP. In this design, the digital back-ends are placed and routed flat along with the rest of the uncore, which is the region of the chip that is not either of the cores or the L2 cache. Because each lane is independent and interfaces with the rest of the chip through an asynchronous interface, the quality of results is superior to SPLASH2. Timing was closed at a digital back-end frequency of over 750 MHz at the slow corner, corresponding to a maximum fast clock of 3.75 GHz or 7.5 Gbps. In practice this frequency was not achieved because of the channel quality and lack of equalization in the analog IP (see Chapter 4).

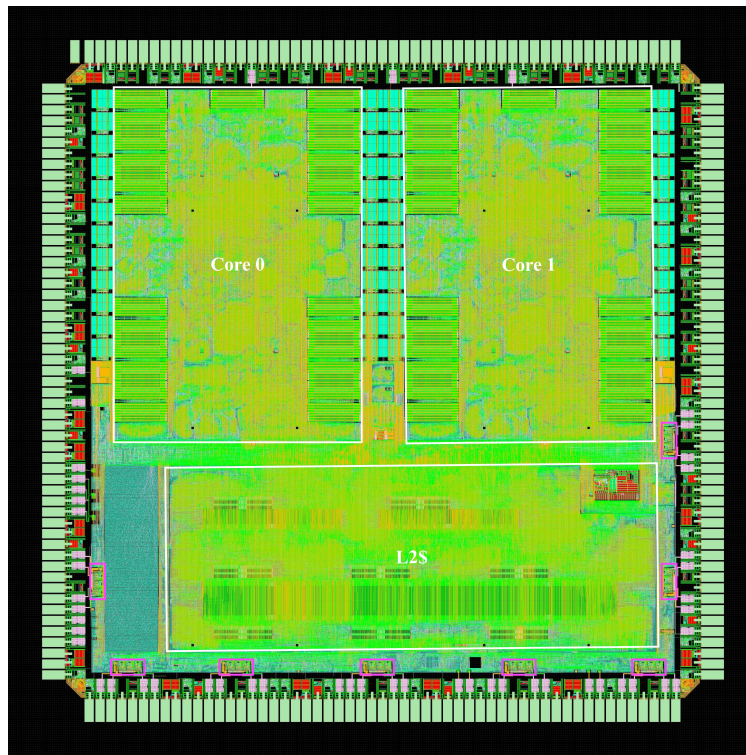


Figure 3.14: Hurricane 1 layout.

The tools were given a distinct fast and slow clock per lane, with an instance of the clock divider per lane to allow the tool to deskew between fast and slow clocks individually. Each fast clock is a clone of the master fast clock, which is the output of a clock multiplexer that selects from the two on-chip LVDS receivers, a single-ended clock receiver, and the on-chip PLL output. Allowing the tool to place these clocks played a role in improving the quality of

results. However, the blockage caused by the L2 placement resulted in routing congestion—a problem that could be fixed with additional floorplanning cycles.

## 3.3 Hurricane 2

### 3.3.1 High BandWidth InterFace (HBWIF) v2.0

#### Organization

In version 2.0, HBWIF (now with the Hurricane prefix dropped) was separated into its own repository for reusability. The links exist separately, and a chip-level configuration is used to assemble the two parts. HBWIF v2.0 was designed using an alpha version of Chisel3: <https://github.com/freechipsproject/chisel3>.

#### TileLink Switcher

TileLink [20] is an on-chip interconnect protocol used by Rocket to drive memory traffic through the cache hierarchy. HBWIF v2.0 supports TileLink natively instead of NASTI. Hurricane 2 leverages this by including a programmable TileLink crossbar called the TileLink switcher (Figure 3.15), which allows the user to select how many HBWIF lanes to use and customize the memory channel mapping. Each HBWIF lane may provide traffic for one or more memory channel, while each memory channel will map directly to a single lane. The TileLink switcher also includes a port for the LBWIF (Low BandWidth InterFace), which is a slow CMOS interface used to bring up the chip before the serial links are active. Configuring the switcher to use this port is handled in the same way as configuring the HBWIF mapping. The TileLink switcher settings are memory-mapped control registers that may be written to using LBWIF commands or by a program running on the core.

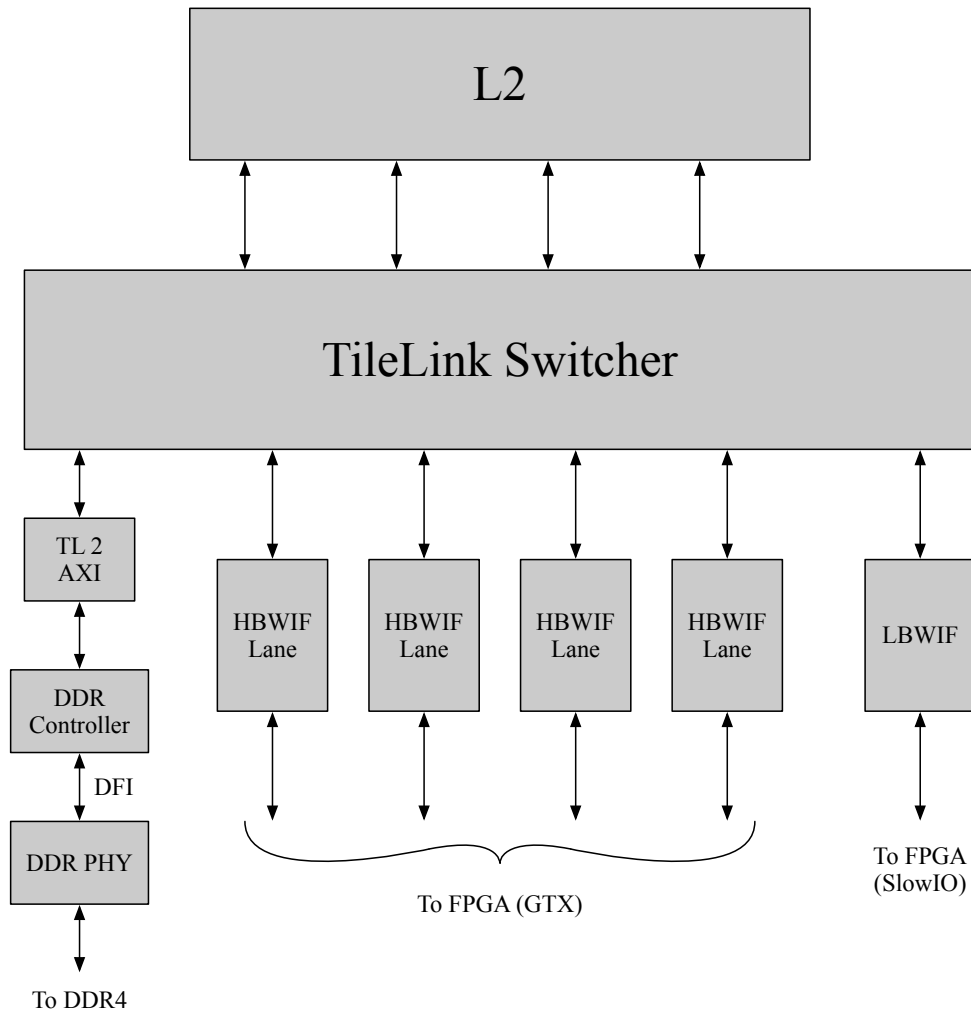


Figure 3.15: Hurricane 2 TileLink switcher.

### Lane Hierarchy

HBWIF v2.0 has more explicit hierarchy within a single lane than the previous generation. Each lane can be divided into three constituent parts based on clock domain: the analog transceiver IP, the digital lane back-end, and the asynchronous crossings. The analog IP is the only block which receives the data-rate fast clock. As mentioned in section 2.4, the slow clock is generated by the analog IP. This clock becomes the source clock for the entirety of the digital back-end, and the lane-side (response enqueue and request dequeue) ports of the asynchronous crossings. The uncore-side of the asynchronous crossings (response dequeue and request enqueue) are driven by the uncore's clock. This organization improves the

readability of the Chisel code, as the clock override is only needed in a single instance.

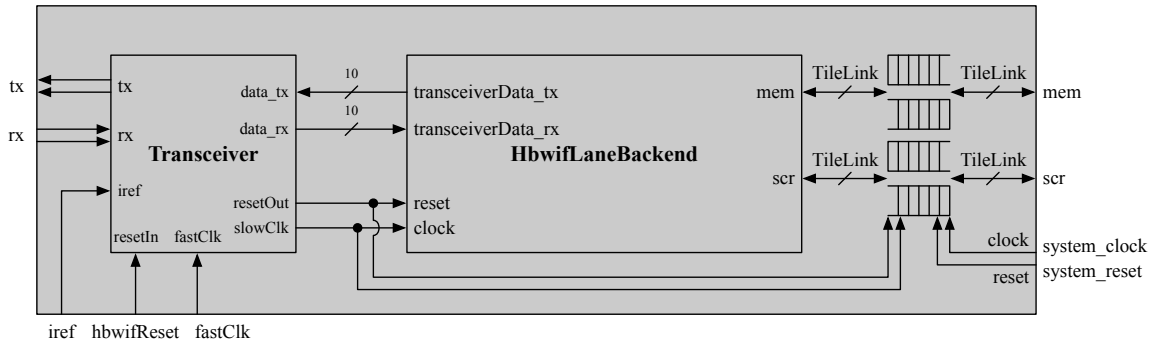


Figure 3.16: Hurricane 2 HBWIF lane.

The digital back-end (HbwifLaneBackend) contains a BERT for testing, a local SCRFile for writing configuration registers and reading status registers, and the primary datapath block; This block is called HbwifTileLinkMemSerDes in Hurricane 2. There is a simple multiplexer to select between the BERT and memory modes.

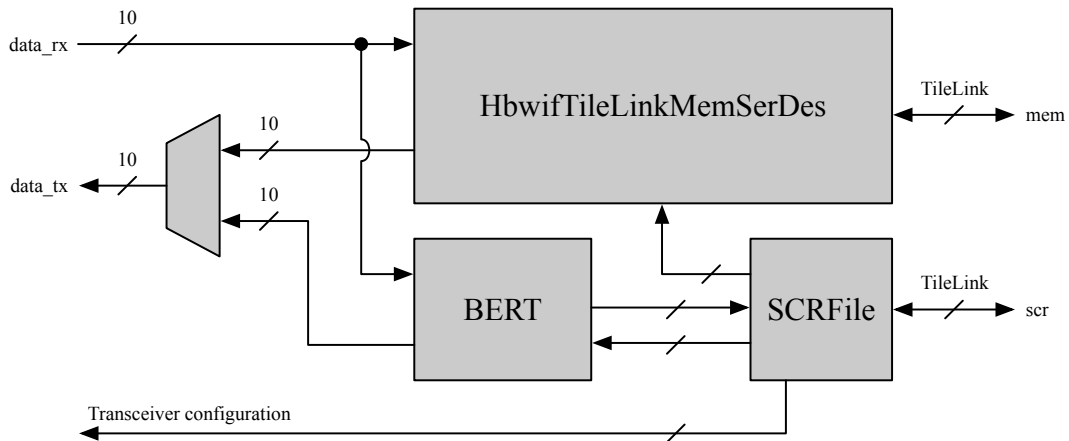


Figure 3.17: Hurricane 2 HBWIF lane back-end (HbwifLaneBackend).

The HbwifTileLinkMemSerDes is similar to what existed in Hurricane 1, sans serializ-er/deserializer and clock divider, which are incorporated into the analog IP. Notable additions are the CRC generator/checker and the retransmit feature, and names of the component blocks are also updated to reflect the TileLink protocol. Figure 3.18 contains a schematic of the HbwifTileLinkMemSerDes.



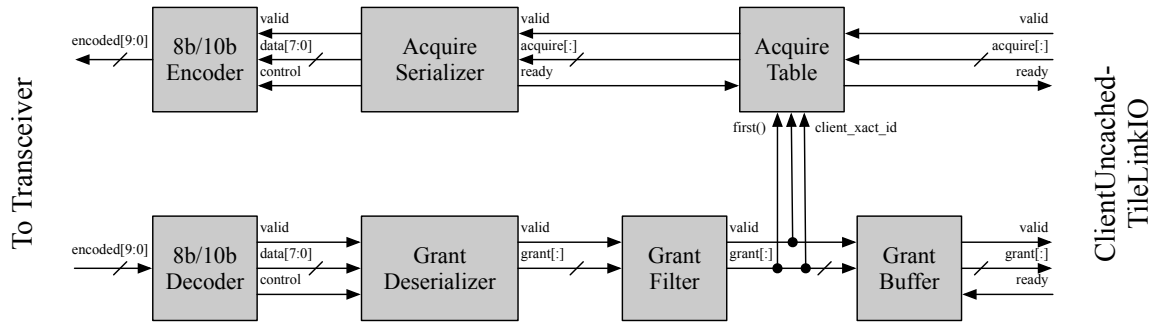


Figure 3.18: Hurricane 2 HBWIF lane micro-architecture (HbwifTileLinkMemSerDes).

The retransmit feature will resend a memory request if a response is not received within a programmable number of cycles. This feature is meant for unreliable channels to improve resiliency, but is disabled by default and enabled by writing to a specific SCR. When enabled, the Acquire table will store a local copy of every memory request along with a timestamp. Each cycle these timestamps are decremented; when zero is reached, the memory request is attempted again. With each response, the ID of the transaction is passed to the Acquire table and the entry is removed.

The Acquire serializer takes a TileLink Acquire (uncached memory request) and breaks it into bytes of data. Each byte of data is passed to the 8b/10b encoder which directly drives the TX data input of the transceiver. After each acquire is sent, a CRC checksum equal to the modular sum of all bytes transmitted is sent. The Acquire deserializer (not pictured) on the FPGA will check this checksum and drop any transaction that fails. Such dropped transactions would be retransmitted once the timeout is reached.

The Grant deserializer will receive bytes of data from the 8b/10b decoder and attempt to construct a TileLink Grant (uncached memory response). Like the Acquire channel, a CRC is sent after the data bytes and is checked by the Grant deserializer. Any incorrect CRC will result in a dropped Grant; causing the transaction to timeout. This will result in the memory transaction being performed again, which is acceptable because the memory system will not attempt to modify data in a location until the previous memory request has been acknowledged.

Once a successful Grant is constructed, the Acquire table is signaled to drop the entry

as mentioned above, and the Grant is entered into the Grant buffer. This buffer is designed to be as large as needed to store all responses from the maximum number of Acquire table entries. The number of entries should not exceed the number of L2 transactors, as this will result in storage that cannot be filled, wasting area.

The FPGA code that cooperates with this circuit behaves in nearly the same manner, except that retransmission is not needed on the FPGA. There is therefore no equivalent to the Acquire table; only a Grant serializer, Acquire deserializer, and Acquire buffer are needed. The 8b/10b principle of operation is unchanged from Hurricane 1.

### DDR4 Comparison

Hurricane 2 contains a DDR4 controller and PHY graciously provided by Cadence Design Systems. Presence of this controller and the serial links will allow a true performance comparison between the two methods. From this, a model will be built to extrapolate performance on chips which do not have a DDR4 PHY but have serial links. Figure 3.19 shows the incorporation of the DDR4 PHY into the Hurricane 2 Chisel code. A Chisel BlackBox wrapper around both the Cadence-supplied controller RTL (written in Verilog) and the PHY was created, and the AXI and AHB ports were tied into the TileLink crossbar using protocol converters. Additionally, SCR values were allocated to configure settings not covered by the controller’s AHB configuration interface.

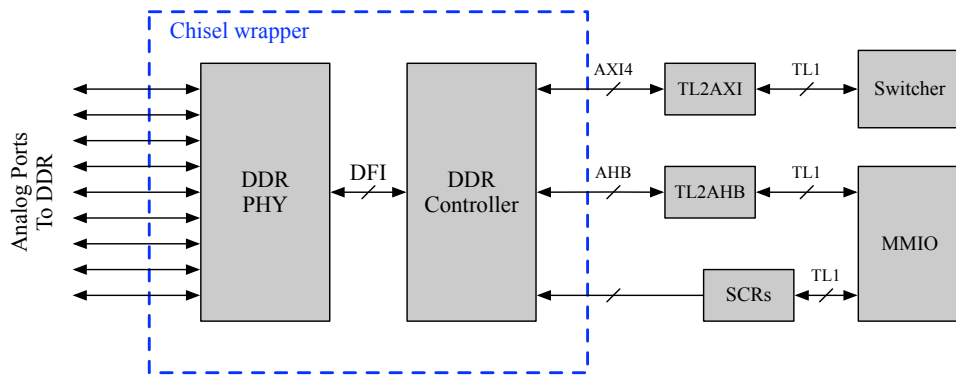


Figure 3.19: Hurricane 2 DDR4 controller wrapper.

### 3.3.2 Physical Design

A GDS of Hurricane 2 is shown in Figure 3.20. Hurricane 2 consists of a single Rocket core with a two-lane Hwacha vector accelerator, a 256 KiB L2 cache, a DDR4 PHY, and 8 HBWIF lanes. Of these, Hwacha, Rocket, and the HBWIF lanes were implemented as hierarchical blocks which were individually placed and routed before being placed inside the top level of the chip. This strategy is helpful for many reasons, including improving clock tree synthesis, shortening tool runtime, and reducing routing complexity.

The HBWIF lanes are indicated as magenta rectangles in Figure 3.20. In addition to being a hierarchical cell, each HBWIF lane is a multiply instantiated module (MIM), meaning that they are identical copies of each other. This reduces the system complexity and allows the individual lane to be optimized before being stamped many times on the chip. Use of this technique further improved the quality of results from Hurricane 1, and led to a chip that met the 1 GHz timing target for the digital back-end.

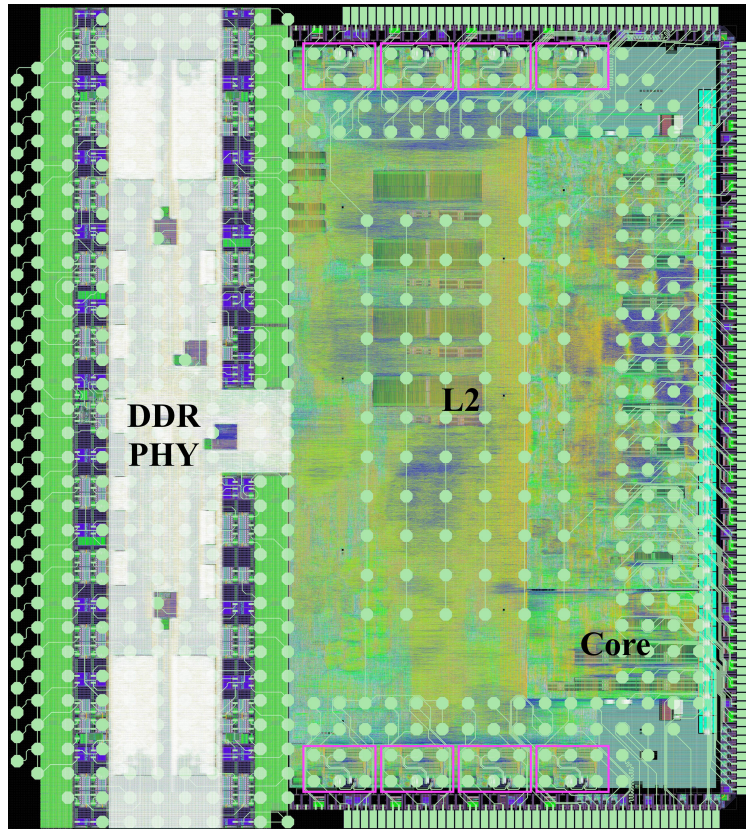


Figure 3.20: Hurricane 2 layout.

The floorplan of a HBWIF lane is shown in Figure 3.21. Each HBWIF lane contains a single transceiver IP and SRAMs used for the Acquire table and Grant buffer. In Figure 3.21 the transceiver IP is located in the top right corner; the remaining boxes are SRAMs. Pins are located on the bottom edge of the lane only, as this block is tiled with other lanes abutting to the left and right at the top level. Custom RDL routing to the bumps is included within the lane, but not shown in Figure 3.21. This custom routing ensures consistent, matched-length traces.

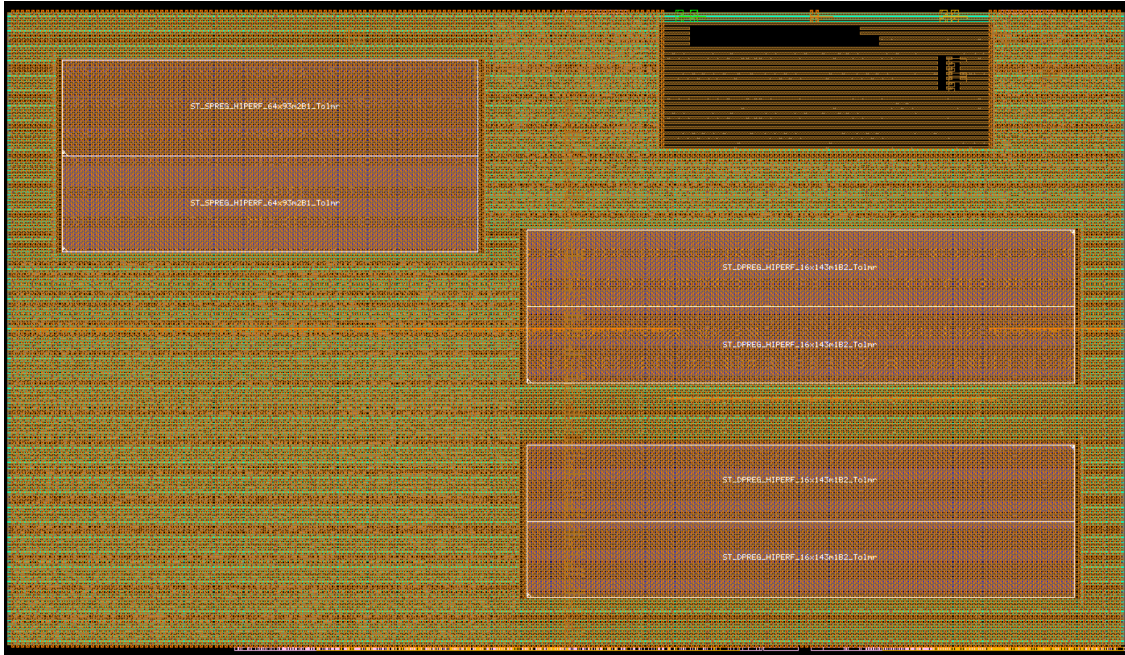


Figure 3.21: Hurricane 2 HBWIF lane layout.

Hurricane 2 contains two banks of HBWIF lanes: one on the top edge of the chip and one on the bottom. The bottom bank of lanes is mirrored about the X axis to keep the digital pins towards the inside of the chip. Each bank has a chip-level bias current input and 4-output current mirror to supply biases to each lane. Each bank also has a chip-level common-mode voltage input that is connected to each lane. This style of design removes the need to run long analog wires across the chip, which would cause congestion and lead to analog signal integrity issues.

# Chapter 4

## Results

A simulation of the TX eye width and height, using a lumped RC model for the channel, is shown in Figure 4.1. This plot is generated using 50 mV of noise superimposed on the supply.

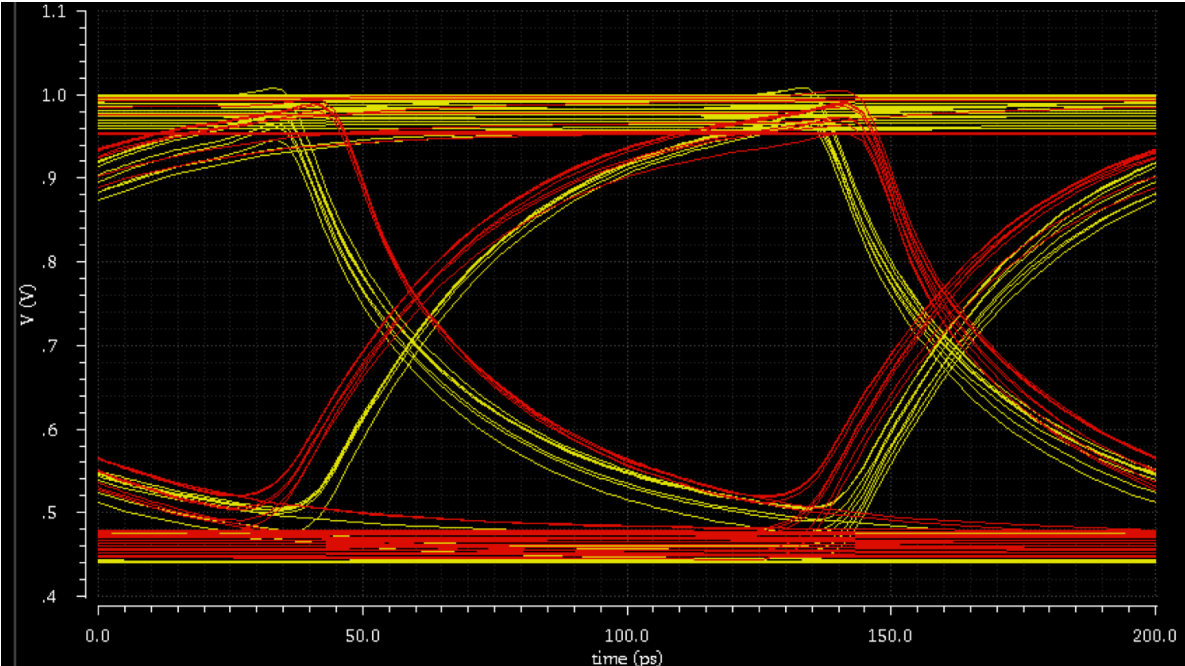


Figure 4.1: Post-layout simulated TX eye diagram (no channel model).

Figure 4.2 demonstrates the movement of memory traffic over the serial links on Hurricane 1. Each yellow bar corresponds to a burst of traffic that is smaller than the timescale of the

image.

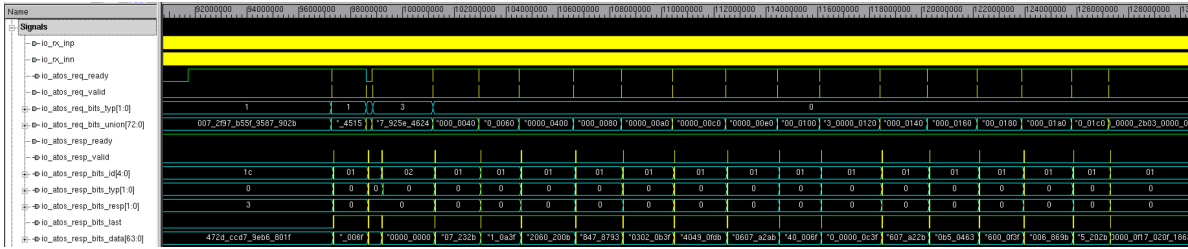


Figure 4.2: Hurricane 1 HBWIF waveforms during an FADD unit test.

Because the on-chip links do not have clock recovery, the phase is adjusted using the Xilinx GTX part’s integrated TX phase interpolators. A calibration routine automatically selects the center of the eye by measuring the bit error rate over a million samples and selecting the phase interpolator code in the center of the eye. The phase interpolators are 7-bit (a maximum of 127) and express a small amount of nonlinearity, as shown in Figure 4.3.

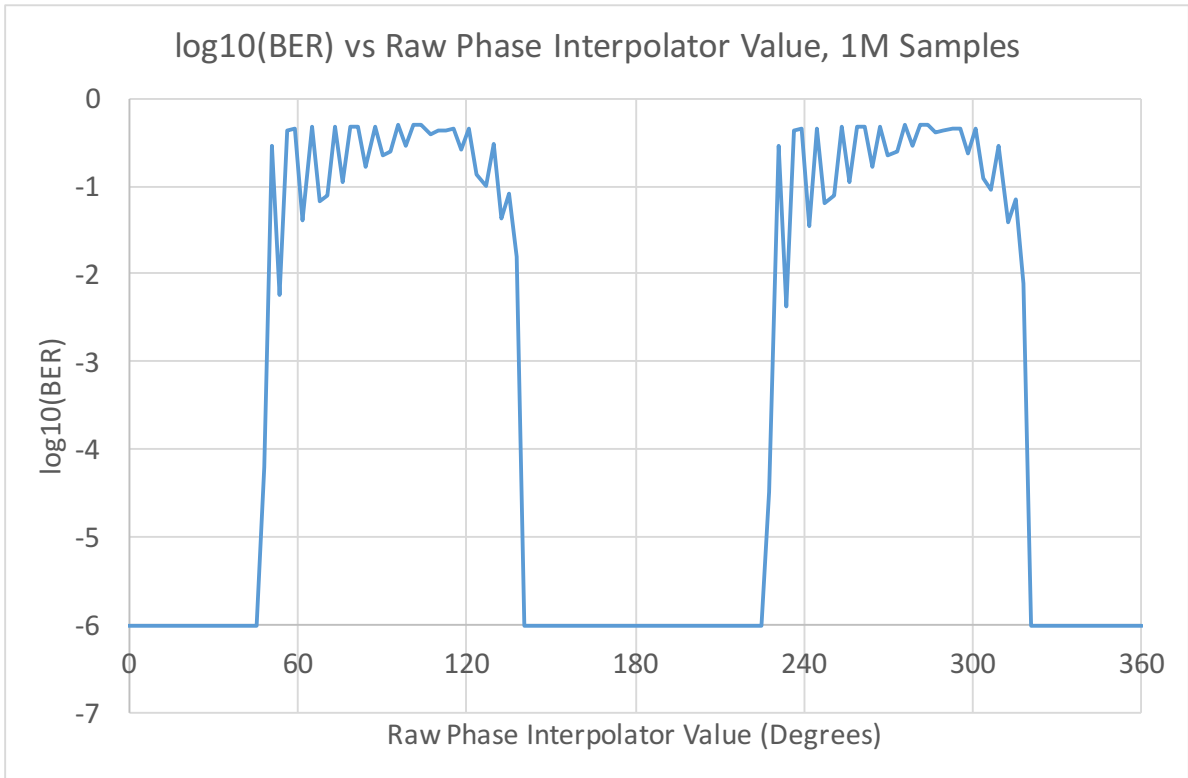


Figure 4.3:  $\log_{10}(BER)$  plot versus FPGA TX phase interpolator value (1M samples).

# Chapter 5

## Conclusion

In summary, a lightweight serial link interface generator has been designed. This includes the custom design of a CML, dual data rate serial link IP as well as a Chisel-based RTL generator. Using this generator, 3 chips of varying complexity have been taped out in STMicroelectronics 28nm FDSOI technology.

Generator-based design is instrumental in accelerating the design of prototypes and test chips. This approach has enabled teams of four to eight graduate students were able to tape out the three chips described in this report in less than a two year time span. This work will help future teams to create chips with higher memory bandwidths at very low cost, enabling future architecture and circuit innovations and providing reasonable data points for performance extrapolation.

### 5.1 Future Work

An improvement to this work is a generator-based approach to the frontend design. This would facilitate reusability and accelerate the design cycle of our test chips. There is a staggered effort to produce serial link generators using the Berkeley Analog Generator (BAG) [21] framework. We have plans to integrate this serial link generator with the back-end

generator to produce a complete subsystem generator for improved automation.

Testing of this system is ongoing. We expect Hurricane 2 silicon in the Fall of 2017, which will allow us to compare the latencies of our approach to an actual production DDR interface to determine the efficacy of this work. While this work is not intended to compete with production-quality interfaces on power or performance, we expect that the results of this work will be adequate to extrapolate the performance of our test chips to full-scale designs., we expect that the performance of this work will be adequate to extrapolate the performance of our test chips to full-scale designs.



# Bibliography

- [1] B. Nikolić, J. Bachrach, E. Alon, K. Asanović, and D. Patterson, “Specialization for energy efficiency using agile development,” in *2015 Fourth Berkeley Symposium on Energy Efficient Electronic Systems (E3S)*, Oct 2015, pp. 1–2.
- [2] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, “The rocket chip generator,” *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
- [3] Y. Lee, B. Zimmer, A. Waterman, A. Puggelli, J. Kwak, R. Jevtic, B. Keller, S. Bailey, M. Blagojevic, P. F. Chiu, H. Cook, R. Avizienis, B. Richards, E. Alon, B. Nikolić, and K. Asanović, “Raven: A 28nm risc-v vector processor with integrated switched-capacitor dc-dc converters and adaptive clocking,” in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–45.
- [4] C. Sun, M. T. Wade, Y. Lee, J. S. Orcutt, L. Alloatti, M. S. Georgas, A. S. Waterman, J. M. Shainline, R. R. Avizienis, S. Lin *et al.*, “Single-chip microprocessor that communicates directly using light,” *Nature*, vol. 528, no. 7583, pp. 534–538, 2015.
- [5] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, “A 45nm 1.3GHz 16.7 double-precision GFLOPS/W RISC-V processor with vector accelerators,” in *ESSCIRC 2014 - 40th European Solid State Circuits Conference (ESSCIRC)*, Sept 2014, pp. 199–202.
- [6] C. Ding and K. Kennedy, “The memory of bandwidth bottleneck and its amelioration by a compiler,” in *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, 2000, pp. 181–189.

- [7] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg, “Data and memory optimization techniques for embedded systems,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 6, no. 2, pp. 149–206, Apr. 2001. [Online]. Available: <http://doi.acm.org/10.1145/375977.375978>
- [8] L. Thayer, “High speed interface for dynamic random access memory (DRAM),” Oct 2013, uS Patent 8,554,991. [Online]. Available: <https://www.google.com/patents/US8554991>
- [9] *Interlaken Protocol Definition*, Cortina Systems and Cisco Systems, Oct 2008, rev 1.2. [Online]. Available: [http://www.interlakenalliance.com/Interlaken\\_Protocol\\_Definition\\_v1.2.pdf](http://www.interlakenalliance.com/Interlaken_Protocol_Definition_v1.2.pdf)
- [10] *Aurora 8B/10B Protocol Specification*, Xilinx, Oct 2014, v2.3. [Online]. Available: [https://www.xilinx.com/support/documentation/ip\\_documentation/aurora\\_8b10b\\_protocol\\_spec\\_sp002.pdf](https://www.xilinx.com/support/documentation/ip_documentation/aurora_8b10b_protocol_spec_sp002.pdf)
- [11] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Aviienis, J. Wawrzynek, and K. Asanovi, “Chisel: Constructing hardware in a scala embedded language,” in *DAC Design Automation Conference 2012*, June 2012, pp. 1212–1221.
- [12] *7 Series FPGAs GTX/GTH Transceivers User Guide*, Xilinx, Dec 2016, v1.12. [Online]. Available: [https://www.xilinx.com/support/documentation/user\\_guides/ug476.7Series\\_Transceivers.pdf](https://www.xilinx.com/support/documentation/user_guides/ug476.7Series_Transceivers.pdf)
- [13] *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide*, Xilinx, Mar 2016, v1.6. [Online]. Available: [https://www.xilinx.com/support/documentation/boards\\_and\\_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf](https://www.xilinx.com/support/documentation/boards_and_kits/zc706/ug954-zc706-eval-board-xc7z045-ap-soc.pdf)
- [14] *3-BIT 26 GSPS ANALOG-TO-DIGITAL CONVERTER W/ OVERRANGE, INHIBIT, AND 1:2 DEMUX*, Analog Devices, v02.0713. [Online]. Available: <http://www.analog.com/media/en/technical-documentation/data-sheets/hmcad5831.pdf>
- [15] T. Kobayashi, K. Nogami, T. Shirotori, Y. Fujimoto, and O. Watanabe, “A current-mode latch sense amplifier and a static power saving input buffer for low-power architecture,” in *1992 Symposium on VLSI Circuits Digest of Technical Papers*, June 1992, pp. 28–29.

- [16] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, W. Hoepfner, D. Kruckemyer, T. H. Lee, P. C. M. Lin, L. Madden, D. Murray, M. H. Pearce, S. Santhanam, K. J. Snyder, R. Stehpany, and S. C. Thierauf, "A 160-MHz, 32-b, 0.5-W CMOS RISC microprocessor," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 11, pp. 1703–1714, Nov 1996.
- [17] D. Schinkel, E. Mensink, E. Klumperink, E. van Tuijl, and B. Nauta, "A double-tail latch-type voltage sense amplifier with 18ps setup+hold time," in *2007 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*, Feb 2007, pp. 314–605.
- [18] N. Mehta, C. Sun, M. Wade, S. Lin, M. Popovic, and V. Stojanovic, "A 12Gb/s, 8.6  $\mu$ App input sensitivity, monolithic-integrated fully differential optical receiver in CMOS 45nm SOI process," in *ESSCIRC Conference 2016: 42nd European Solid-State Circuits Conference*, Sept 2016, pp. 491–494.
- [19] A. X. Widmer and P. A. Franaszek, "A DC-balanced, partitioned-block, 8B/10B transmission code," *IBM Journal of Research and Development*, vol. 27, no. 5, pp. 440–451, Sept 1983.
- [20] H. Cook, "Productive design of extensible on-chip memory hierarchies," Ph.D. dissertation, EECS Department, University of California, Berkeley, May 2016. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-89.html>
- [21] J. W. Crossley, "BAG: A designer-oriented framework for the development of AMS circuit generators," Ph.D. dissertation, EECS Department, University of California, Berkeley, Dec 2014. [Online]. Available: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-195.html>