BRGTC6: Source-Synchronous Parallel Chip-to-Chip Link

Barry Lyu, Parker Schless, Vayun Tiwari
Cornell University, Computer Systems Laboratory
Advisor: Dr. Christopher Batten

I. Abstract

Sending data across two chips reliably involves many challenges, including clock-domain-crossing (CDC), flow control, error detection, etc. There are many different schemes and protocols already in place for data transfer with different trade-offs. Simple protocols such as SPI and I^2C provide light-weight synchronous serial transfers at low frequencies and are ideal for low-data-rate transfers such as debugging or peripheral connections. Advanced protocols such as PCIE or USB use special encodings and clock-data-recovery for high speed serial transfers. They usually require complicated SERDES logic and analog integration, and are less suitable for light-weight chip-to-chip or mesh-on-chip links. Parallel links are alternatives to serial links, although recent serial links have gained popularity as they require fewer I/O pins and are not subject to skew between data lines. Modern parallel links such as DDR (double data-rate) links for DRAM and HBM provide wide and extremely high data-rate links at the cost of complex controllers and training sequences. The goal of BRGTC6 is to find balance between performance, latency, and complexity through a unified, lightweight, parallel link, applicable for chip-to-chip, chiplet, and mesh-on-chip communications. We developed an 8-bit wide, single-data-rate, source-synchronous interface with credit-based flow control, built-in self testing and repair-ability, and taped-out a $1\,mm^2$, $200\,MHz$ test chip in TSMC 65nm (Figure 1) for validation and evaluation, achieving a throughput of $1.6\,Gb/s$ at $200\,MT/s$ in post-silicon testing. [1] Our full architecture is seen in Figure 2.

II. Architecture

The minimal setup required to send data across chips is a source-synchronous interface that forwards the message and valid bits along with the clock. The downstream chip would sample the message relative to the forwarded clock and then employ an asynchronous FIFO for CDC to its own clock domain (Figure 3). This setup, however, requires the downstream link to drain the FIFO faster than the upstream link, otherwise messages might be discarded as the FIFO fills up. It is also vulnerable to skew and may only be able to operate at low frequencies. In later subsections, we detail how we build up from minimal setup to the BRGTC6 link, addressing issues and vulnerabilities.

[1] In STA, our design closed on timing for a clock period of $6\,ns$ ($166\,MHz$). However, in post-silicon testing, our chip was able to run reliably up to $200\,MHz$. We attribute this to conservative margins in our timing constraints, including large clock uncertainties and extra input delays.

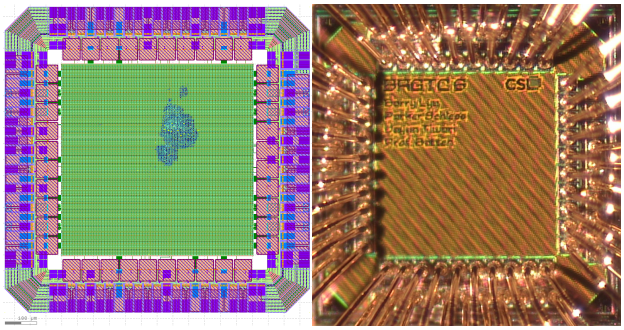

Fig. 1: BRGTC6 GDS (left) and die-shot (right)

A. Link Reset Scheme

Async-FIFOs operate across two clock domains and typically requires two synchronous reset signals, relative to the write clock and read clock respectively. This synchronous reset scheme, however, is not ideal for chip-to-chip links: depending on the power-on and reset sequence, an arbitrary chip might come out of reset first and start operating while the other chip is still in reset: If the receiving chip comes out of reset first, it will enter a metastable state as the write pointer of the chip has not yet been reset.

To account for this issue, we employ a unified reset scheme for the async-FIFO (Figure 4): the upstream chip sends a `link_reset` to the downstream chip. `link_reset` is tied to the upstream chip's core reset (`up_reset`) so they assert together, but `link_reset` de-asserts after a fixed 10 cycle delay to ensure the link is held in reset longer than the chip. The active high `link_reset` asynchronously resets both the write and read pointers of the downstream FIFO when asserted. However, we use a special reset synchronizer to synchronously de-assert it, preventing potential metastability issues on de-assertion. Overall, this scheme guarantees that as long as there is a period of time when both chips are in reset, the sequence in which chips come out of reset does not affect normal functionality.

B. Channel Clock Division

Physical links typically run slower than core logic due to delays from I/O and external wiring. Thus, we want to be able to run links at a lower frequency than the core. In complex links,

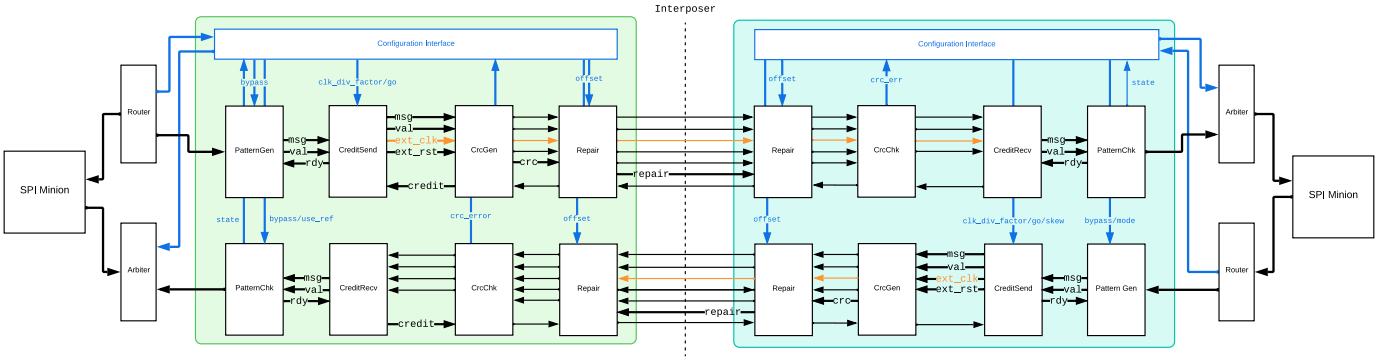

Fig. 2: BRGTC6 architecture

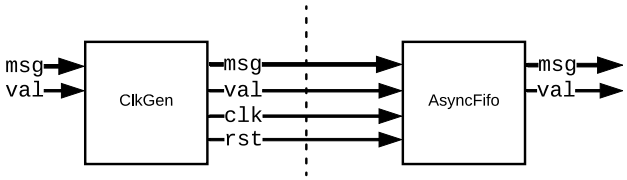

Fig. 3: Minimal chip-to-chip source-synchronous interface

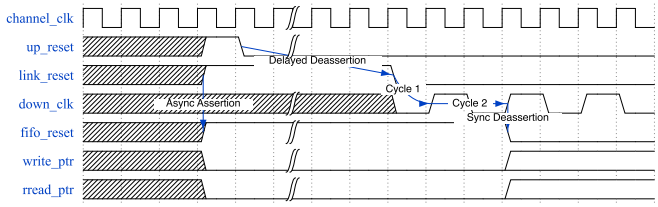

Fig. 4: Async-FIFO reset scheme

this is typically achieved with a PLL (phase-locked loop), but this is a heavy weight solution that typically involves analog integration and adds significant complexities to verification and physical design. As a lightweight alternative, we use a counter-based clock divider to divide the core clock (Figure 3). The divider supports division by arbitrary multiples of 2, and we gate the data lines so data only toggles with the divided clock.

C. Credit-Based Flow Control

To handle downstream back pressure, we included credit-based flow control in our link: upstream chips are assigned a predefined number of credits, according to the depth of the downstream async-FIFO (16 for BRGTC6). The upstream link deducts one credit whenever sending a message, and may only send messages when it has credits remaining. The downstream link, on the other hand, would send one credit back whenever a message is pulled from its async-FIFO, allowing the upstream link to send another message.

While credit-based flow control handles back pressure, returning credits involves sending signals backwards from downstream to upstream. In a two-way link, credit returns can be implemented by embedding credit returns in the opposite channel, so credits returned by a chip will be forwarded

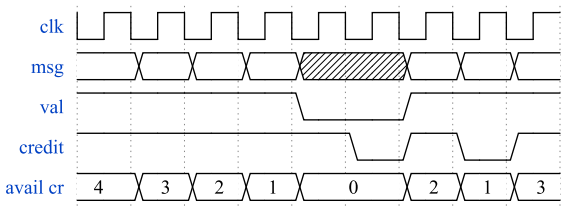

Fig. 5: Example credit return waveform

along with its clock to be sampled by the other chip. To allow for one-way links and decoupling credit returns from data channels, we employed a fully asynchronous credit-return scheme (Figure 5): when the downstream link wants to return a credit, it will toggle the `credit` line of the link. The upstream chip implements an asynchronous counter (equivalent to a 0-bit async-FIFO with only pointer handling logic) and counts the number of posedges on the `credit` line. At every cycle, if the counter is non-zero, the upstream chip decrements the counter and adds 2 to its current number of credits. [2] We bundled clock dividers and credit handling logic into `CreditSend` and `CreditRecv` modules to jointly handle the logic (Figure 2).

D. Configuration Interface

To be able to configure and debug the chip, we include an SPI minion and a configuration interface. The configuration interface has an array of configuration registers, such that an external SPI master[3] can poll status registers to retrieve the status of the chip, or write settings registers to configure the chip. Available configurations include clock division factors, built-in testing settings, skewing adjustments, etc. Statuses include link status, failure count, received messages, etc.

E. Skew Tolerance

For chip-to-chip links, skew between inter-chip lines are hard to predict. Multiple factors can cause skew between data lines, including different wire-bond lengths, PCB traces, and on-chip driving/sampling logic. To account for possible skews,

[2]Since the downstream chip toggles for every 1 credit, encountering a posedge suggests that 2 credits have been returned, so we increment by 2.

[3]For BRGTC6, we use an RP2040 MCU to drive the SPI interface.

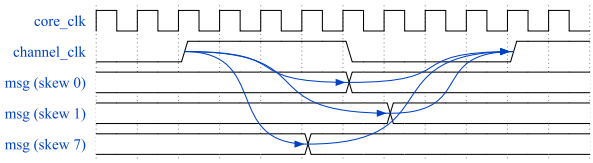

Fig. 6: Skew control mechanism

we toggle data relative to the falling edge of the channel clock, leaving margin for the setup and hold time so downstream can sample data correctly (Figure 6). To allow for more fine-tuned control against severe skews, we added a lightweight mechanism for artificially controlling the relationship between the clock and the data pins. When operating the link based on a divided clock, we maintain a counter off the original clock to track when data can be toggled. By introducing an offset to the counter, we can shift when data toggles relative to the original clock. For example, with a clock divisor of 8, we can adjust the phase of the data by 1/8 of a channel clock period. This adjustment can be manually set through the configuration interface to adjust the setup and hold relationships, as well as the counter skew in the channel.

F. Built-in Testing Functionality

While we could test the link by manually sending messages across the link with the configuration interface, SPI operates much slower than the link and cannot achieve full throughput of the link. To test the link exhaustively at full throughput, we included built-in testing functionalities that generate and send patterns in the upstream and checks against those patterns in the downstream (Figure 2). The `PatternGen` and corresponding `PatternChk` modules support two operating modes: random patterns or alternating fixed patterns. In random testing mode, pseudo-random patterns are generated and checked against a Linear Feedback Shift Register (LFSR). In fixed pattern testing mode, two 8-bit patterns can be set through the configuration interface and the `PatternGen` module will alternate between sending the two patterns. The `PatternChk` module in the downstream chip will check received messages against the patterns. When there is a mismatch, the erroneous message can be polled through the configuration interface to allow for further debugging. For example, when polled messages show that a single bit never toggles, it could be that the pad or connection is dead. Upon initiating a test sequence through the configuration register, patterns will be continuously sent through the link, and then upon receiving 256 correct messages in a row, the link indicates that it is ”locked” through a status register.

G. CRC and Repair

Lastly, to add robustness to the link, we include CRC and repair modules. The CRC unit adds a parity bit to the link, allowing for downstream parity checks to identify failures (the parity bits cannot be used for error correction, however). The repair unit adds a repair line to the link, which allows for shifting I/O signals to another pad to accommodate for any single pad or connection failure. Both modules' repair selection can be configured through the configuration interface.

III. Verification

Pre-silicon verification for our chip-to-chip link is essential to ensuring correct functionality due to the asynchronous nature of the design. While previous BRG tapeouts have been able to use PyMTL to perform verification, it does not support asynchronous inputs and outputs. In addition, PyMTL is designed to be used for verification of a single hardware block, whereas we want to be able to test two of our links communicating in simulation. To accomplish this, we use a pure Verilog testing methodology that allows us to have complete control over all testing elements, while also allowing for robust controlled-randomization support and module parameterization.

A. Testbench Structure

Our integration tests are set up using the testbench structure seen in Figure 7. Two instances of our link are instantiated, along with automated sources and sinks that send and receive test stimuli to and from the DUT's. A centralized `run()` task executes each test case which controls the messages to be sent and received by the sources and sinks. Multiple such testbenches can be instantiated via parameterization so that DUT's with different parameters such as different bitwidths can be tested. Since each link interfaces with the test environment through SPI signals, we have created a testbench element `SpiDriverValRdy` which handles converting the val/rdy microprotocol of the test sources and sinks into SPI signals for communicating with the links.

Since a key part of our design is the ability to handle clock skewing on the link, we need to be able to insert such delays as part of our test bench. To do this, the `CredSkewer` module is used which inserts configurable, artificial delays into each bit of the link using shift registers whose lengths are able to be set as part of the specific test case. Additionally, errors can be injected into each bit of the link to test error checking capabilities.

B. Integration Tests

Our integration tests are designed to test as many aspects of the links as possible to ensure robust communication in all scenarios. Such tests exercise the communication functionality for various clock periods and clock period differences between chips, different amounts of skew on the credit interface by setting the `CredSkewer` modules appropriately, and different settings of the credit clock generator. Back-pressure tests are also performed where the testbench will control exactly when messages on either side of the link are sent and/or received to ensure no message is lost from end-to-end when internal backups occur. Our links are capable of sending a variety of message types, including fixed patterns, random patterns generated by an LFSR, and channel messages sent via the SPI interface. All of these types of messages are tested under the various conditions listed above.

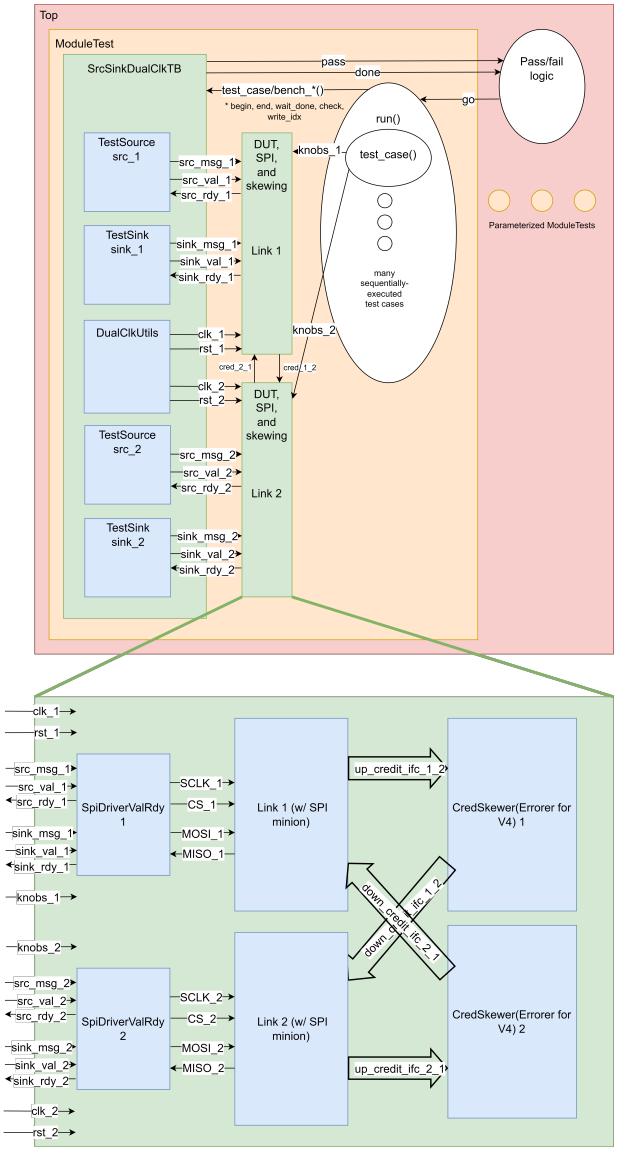

Fig. 7: Testbench structure and module connections for BRGTC6 dual-chip integration tests

C. Constrained Randomization

In PyMTL testing, randomization is performed by globally setting a seed in the testbench. While ideal for reproducibility since the same set of messages are produced every time in the same order, such a simple mechanism is not ideal for our application as we want to be able to randomize our many testbench parameters with as many combinations as possible to ensure robustness. To fix this, we use the DPI interface to pass in a randomly-generated seed via a C program which obtains the system time. Using the system time as the seed means that we will be guaranteed a different seed for every run of the program since it counts the number of seconds since a specific date in history. To implement constrained randomization of the test case parameters, this seed is added to the test case number and passed to the test case function itself, thus making each seed different for each test case within the simulation. A number of optional parameters are also available to be passed into each test case, such as the skew amounts for the `CredSkewer`, individual clock periods, reset delays, repair bit selection, etc. If these optional parameters are not explicitly provided by the user, then they are randomized according to the seed and pre-defined minimum and maximum bounds. Additionally, the seed for each test case is printed out when the test is run, so a specific seed for a specific test case and be explicitly added to the run list so that that series of messages and parameters can be tested every time the test bench is run.

D. Testing Modes

Our RTL simulations support both Verilator and VCS. We also include support for FFGL simulation and BAGL simulation for our integration tests with VCS to ensure proper functionality post-synthesis and post-place-and-route, respectively.

IV. ASIC Setup

Our PDK (Process Development Kit) provider is MUSE (Multi-Project-Wafer) University Service. From them, we were provided with `.tar.gz` archive files, which contain all necessary dependencies for a tapeout. In the Batten Research Group, we split the PDK into two categories: (1) the 'PDK' which contains device models and is used for full custom designs, and (2) the 'ADK' (or ASIC Design Kit), which contains dependencies for standard cell designs. For BRGTC6, we focus on the ADK.

Using a shell script, we untar all of the directories in the ADK, and create a standard view. Since there are many files, with complicated directory structures, this is a specific design methodology to keep an organized list of all files that our tooling scripts point to.

Our ASIC flow is implemented and automated with CMake. The high-level flow consists of the following steps:

1) *Initialization:* Copies and modifies the ADK standard view files, setting up the necessary directory structure and preparing the environment for the subsequent flow stages.
2) *Pickling:* Combines all source Verilog files and include paths into a single file using Verilator. This process ensures that all modules are explicitly defined within a single file, making it suitable for synthesis.
3) *Synthesis:* Runs Synopsys Design Compiler with the provided synthesis scripts to generate a synthesized netlist, SDC, SDF, and timing reports based on the pickled Verilog source.
4) *FFGL simulation:* Performed post-synthesis using VCS to validate the synthesized netlist against the expected functionality. The primary focus in this stage is to verify that the synthesized gate-level netlist behaves identically to the RTL design.
5) *Place and route (PnR):* Executes the place and route flow using Cadence Innovus. This step includes IO assignment generation, floorplanning, power routing, clock

tree synthesis, and routing to produce the post-layout GDS and timing information.

6) *BAGL simulation:* Performed post-PnR using VCS to verify the timing and functionality of the layout netlist. This simulation incorporates extracted parasitic data (SPEF) and the final SDF generated during PnR to accurately model delays and coupling effects.
7) *Merge watermark:* Merges a custom watermark GDS with the PnR output GDS using Mentor Calibre.
8) *Merge sealring:* Integrates the ADK-provided sealring GDS with the watermarked GDS using Calibre, forming a complete top-level sealringed layout.
9) *Generate and insert dummy filler cells:* Creates dummy filler cells using Calibre to fill gaps in the layout and merges the filler GDS with the sealringed GDS, producing the final filled layout.
10) *Design rules check (DRC):* Runs DRC on the filled GDS using Calibre to verify that the layout complies with the PDK-specific design rules for metal spacing, width, and other layout constraints.
11) *Layout vs. schematic (LVS):* Performs LVS to compare the final layout against the synthesized netlist, ensuring that the physical layout matches the intended circuit design. This is done using Calibre with hierarchical cell handling to isolate standard cells and reduce false error checks.

V. Synthesis & Timing Constraints

The synthesis flow for BRGTC6 is managed through a comprehensive TCL script in Synopsys Design Compiler, which defines the key timing constraints necessary for ensuring functional integrity across the core, link, and credit domains. Given the asynchronous nature of several control paths and the presence of multiple clock domains, precise timing constraints are essential to prevent false path optimizations and misaligned data transfers.

A. Clock Definitions

Two primary clocks and one generated clock are defined in the synthesis script to represent the different timing domains in BRGTC6:

- Core clock (`clk_core`): defined with a period of 6 ns, corresponding to a nominal frequency of $166\,MHz$. It is sourced from the `clk_pad` input and drives the main datapath and logic throughout the design.
- Downstream credit clock (`down_cred_clk`): operates at the same frequency as the core clock but is treated as an independent clock domain to account for asynchronous credit returns. To account for wiring delays, the clock is constrained with some clock uncertainty.
- Upstream credit clock (`up_cred_clk`): produced by dividing the core clock by a factor `clk_div_factor`. A value of 1 disables division while larger values (2, 4, 8) allow for programmable skew control. The clock is also inverted, placing the rising edge of `up_cred_clk` exactly half a channel-clock period after the core-clock rising edge, so data launched on the core clock's falling edge reaches the receiver *before* the sampling edge, as explained above.

B. Input and Output Delays

To ensure accurate timing analysis across the chip-to-chip link, both input and output delays are explicitly defined relative to the channel clocks. The `PDDW12DGZ_G` pad cell is declared as the external driver for all inputs or load of all output ports to accurately model RC strength, slew rates, and buffer delays. This standardizes the delay calculation and mitigates discrepancies introduced by varying drive strengths.

a) Input Delays: Input delays are defined relative to the `down_cred_clk`, accounting for potential skew and trace mismatch. The delay window is calculated as:

$$\begin{aligned}\text{max\_input\_delay} &= T_{\text{clk}} - \text{down\_cred\_clk\_margin} \\ &= 6\,\text{ns} - 2\,\text{ns} = 4\,\text{ns}\end{aligned}$$

$$\text{min\_input\_delay} = \text{down\_cred\_clk\_margin} = 2\,\text{ns}$$

Data can therefore arrive anytime within a 2 ns to 4 ns window relative to the sampling edge of `down_cred_clk`.

b) Output Delays: Output delays are defined relative to the `up_cred_clk`, accounting for clock inversion and data launch timing. The delay window is calculated as:

$$\begin{aligned}\text{max\_output\_delay} &= \frac{T_{\text{clk}}}{2} - \text{up\_cred\_clk\_margin} \\ &= 3\,\text{ns} - 2.75\,\text{ns} = 0.25\,\text{ns}\end{aligned}$$

$$\begin{aligned}\text{min\_output\_delay} &= \text{up\_cred\_clk\_margin} - \frac{T_{\text{clk}}}{2} \\ &= 2.75\,\text{ns} - 3\,\text{ns} = -0.25\,\text{ns}\end{aligned}$$

This configuration centers the sampling edge in the middle of the data valid window. Both input and output delays use `source_latency_included` and `network_latency_included` to capture on-chip routing and pad delays, ensuring that timing analysis accounts for all possible data launch and arrival scenarios.

C. Asynchronous Paths

Asynchronous paths involve signals that are not launched and sampled based on the same clock edge. We use the `set_false_path` command to exclude these paths from timing analysis. This prevents Design Compiler from attempting to optimize or constrain them, as they do not have deterministic timing relationships.

- Reset path (asynchronous reset synchronizer): from external reset input to synchronizer register input
- SPI interface: from SPI clock, MOSI, chip select to synchronizer registers within SPI minion
- Upstream credit send counter and downstream credit receive FIFO: all paths to read and write point synchronizers.
- CRC: input to the synchronizer register for CRC error flags.

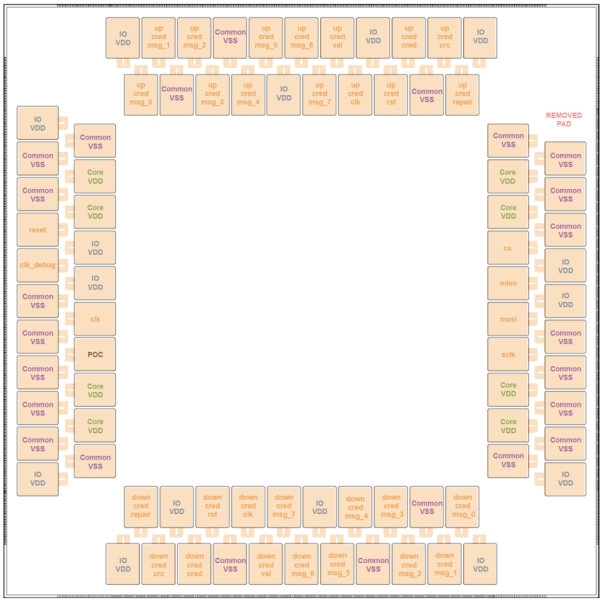

Fig. 8: BRGTC6 staggered padring and pinout

We stream out Synopsys Design Constraints (`.sdc`) for use in Place & Route and Standard Delay Format (`.sdf`) for use in VCS.

VI. Physical Implementation

A key challenge in the physical implementation of BRGTC6 was effectively managing a high signal count within a limited die perimeter. There are 35 data signals, 16 IO VDD, 8 core VDD, 24 common GND, and 1 power-on-control, totaling at 84 required I/O. Due to the sealring and corner cells, our effective side length was shrunk to 700 microns.

CPGA wire-bonding packaging constraints required bond-pad pitch to be greater than 60 microns. Around the total perimeter, this limits us to around 46 bondpads: not nearly enough for our purpose.

We decided upon a staggered bondpad scheme (Figure 8), which is possible because our ADK defines I/O cells and bond-pads separately. Using the circuit-under-pad (CUP) scheme, each metal bondpad requires one I/O cell (which handles signal driving to or from the core) to be placed beneath it. The I/O cells are placed in-line along the perimeter of the die, and pads are placed with alternating orientations, creating an inner and outer ring of pads. While this scheme reduces the core logic area, it effectively doubles the amount of signals one can route on/off the chip. As seen in the GDS (Figure 1, this was not an issue, as the total amount of logic for our link was small relative to our available area.

VII. Post Silicon Validation

Previous validation solutions in the group involved using bulky lab equipment and complicated SPI drivers to write SPI transactions one at a time to a single device. For our tapeout, being able to communicate with two chips at once and with

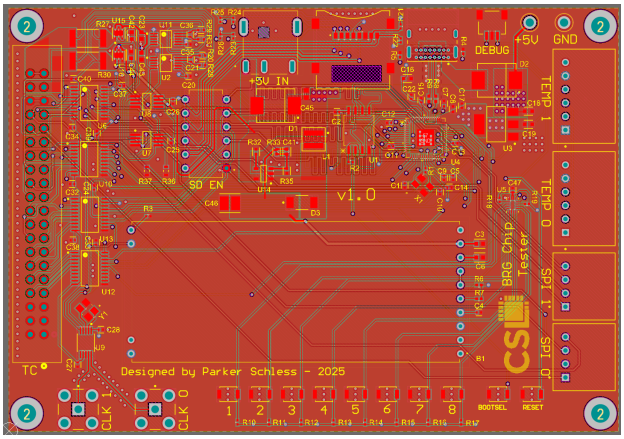

Fig. 9: Chip Tester board layout

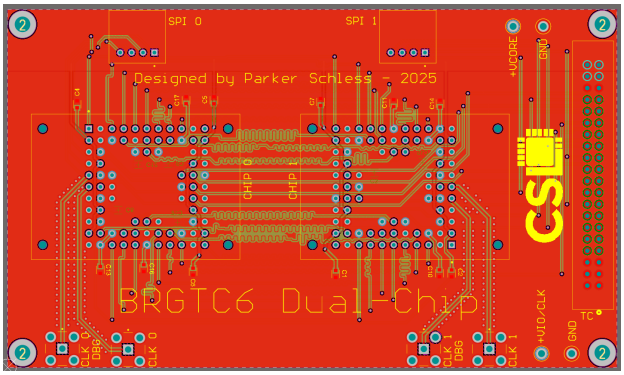

Fig. 10: BRGTC6 dual-chip daughter board layout

fast transactions was paramount to effective testing. Additionally, we wanted to make a platform suitable for demoing the chip. To accomplish this, we developed a system utilizing a central Chip Tester board (Figure 9) as well as a dedicated daughter board that implements the traces connecting each chips' links (Figure 10). The daughter board is then able to communicate over a standard 40-pin header to the Chip Tester (Figure 11) which uses an RP2040 microcontroller to drive the two SPI busses at once. The Chip Tester also includes on-board LDO's which generate the core and I/O voltages and can be configured via the RP2040. A configurable clock generator is also on the board which can generate two independent clock frequencies up to 200 MHz. Buttons and an OLED screen on the board can be used to start and run tests/demos independently, but the board can also be connected to a computer over a standard USB-C connection to send and receive commands. In addition to a dual-chip daughter board, a single-chip daughter board has also been populated which acts as a standard breakout board for all pins of the chip. Such a board is used for testing a chip in isolation, as well as future emulation with an FPGA.

VIII. Evaluation

To evaluate our dual-chip setup, we vary the clock frequency of each chip with a nominal I/O voltage of 3.3V and nominal

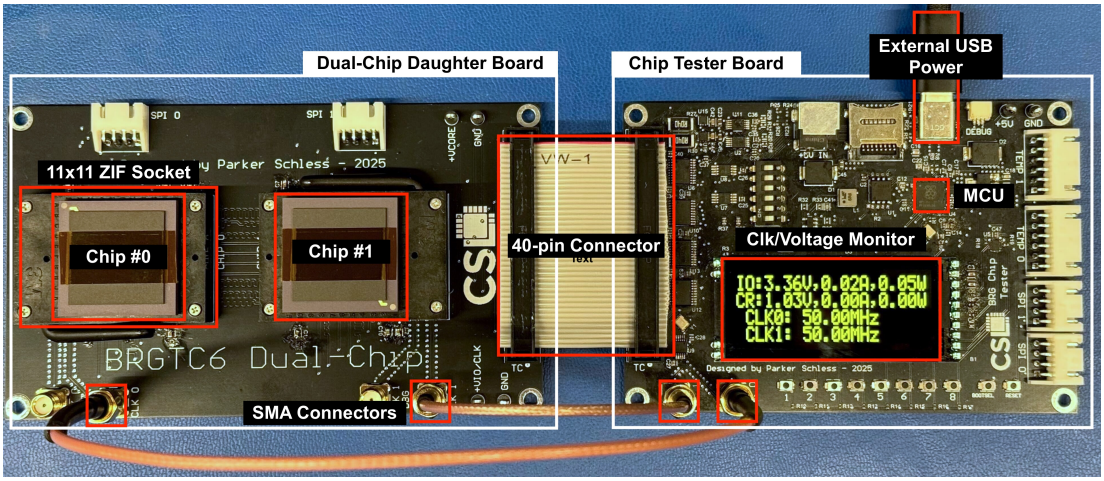

Fig. 11: Testing setup

Parameter	Value
Clock Divider Factor	1
Clock Divider Skew	0
Pattern Mode	0 (LFSR)
Pattern Bypass	0 (no bypass)
Up Repair Select	0 (no usage)
Down Repair Select	0 (no usage)

TABLE I: Configuration parameters for each chip for the frequency sweeps. Both chips are set with the parameters above.

core voltage of 1.0V from $150\,MHz$ to $250\,MHz$ on each chip. The parameters of the chips during the test are tabulated in Table I. After setting those parameters, a "go" signal is asserted on both chips, and both chips send random messages using the LFSR for 30 seconds before the Pattern State register is read again, upon which a reading of 2 indicates that there were no errors. Reading the Pattern Error Count register should also return 0 to indicate there were no errors. The results of the frequency sweeps are shown in Figure 12. As can be seen, for all combinations of frequencies $200\,MHz$ and under, both chips pass. When the upstream chip is configured for $210\,MHz$, the downstream chip fails to receive the correct messages, an interesting result given that the same test at $220\,MHz$ passes on both chips. The reasoning for this phenomenon is unclear, but we suspect it is related to a setup time failure when a different path is triggered for $>200\,MHz$. Further testing is needed to verify this hypothesis. Other than the $210\,MHz$ anomaly, both chips continue to work up until $230\,MHz$ (with the exception of a single chip failure for a $220\,MHz$ test), where both chips start to fail by incurring a nonzero number of errors on the link during the 30 second test duration. Based on this, we can conclude that our link as implemented on our specific dual-chip daughter board has a bandwidth of $BW = 8 \cdot (200e6) = 1.6\,Gb/s$ or $200\,MT/s$ given our link is 8 bits wide.

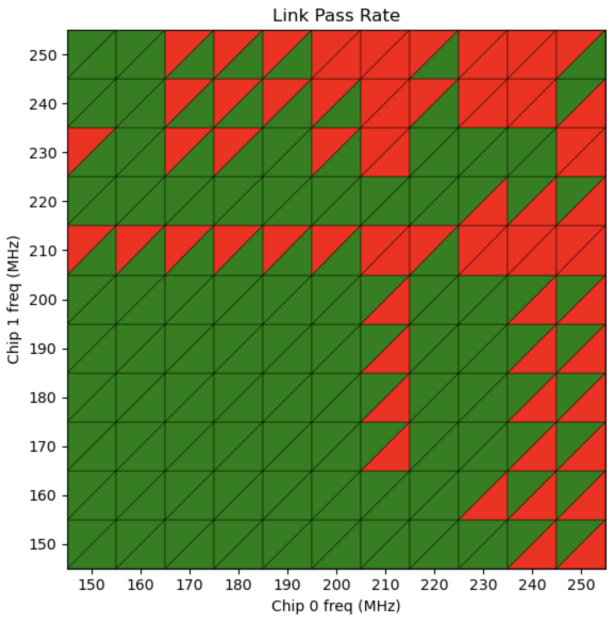

Fig. 12: BRGTC6 frequency sweep tests performed after setting the parameters in Table I to both chips. The links run for 30 seconds after the "go" signal is asserted on both chips. Each grid position is split in half, with the left upper triangle corresponding to chip 1's pass/fail status and the right lower triangle corresponding to chip 0's pass/fail. Red indicates a failure and green indicates a pass.