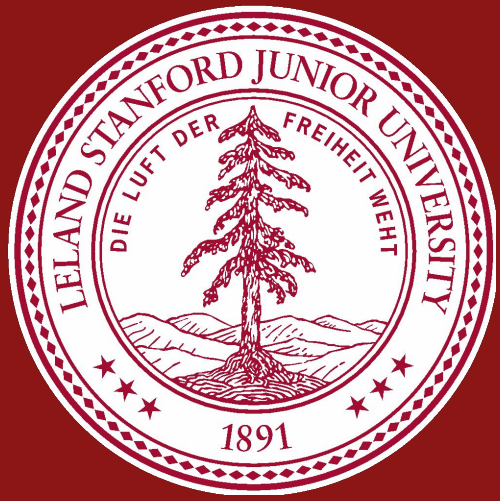




Andrea Giuri
Author · Stanford EE

Integration, Verification, and Optimization of a HyperBus Controller for HyperRAM, SemperFlash, and Chip-to-Chip Interconnect

Andrea Giuri, Christian Kubicka, Thierry Tambe
Department of Electrical Engineering, Stanford University — Tambe Lab



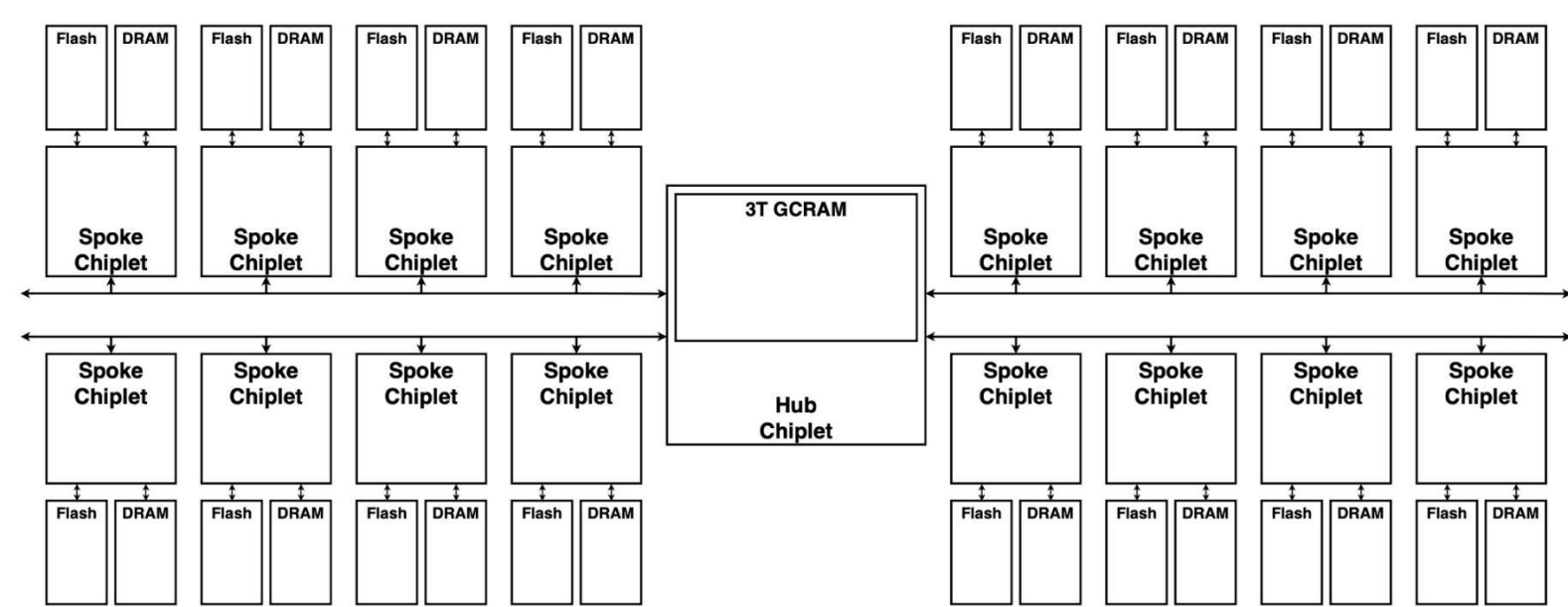
Christian Kubicka
Mentor · Stanford EE



Prof. Thierry Tambe
Principal Investigator

Introduction & Motivation

- **μVLA** is a multi-chip package for low-latency inference of Vision-Language-Action (VLA) models: a **Hub SoC**, several **Spoke SoCs** with AI accelerators, and two **memory chips** per Spoke: one HyperRAM and one SemperFlash.
- Connecting every die requires a fast, low-pin-count interconnect. **HyperBus** was chosen as that shared standard, for both **off-chip memory access** and **chip-to-chip** communication between chiplets.



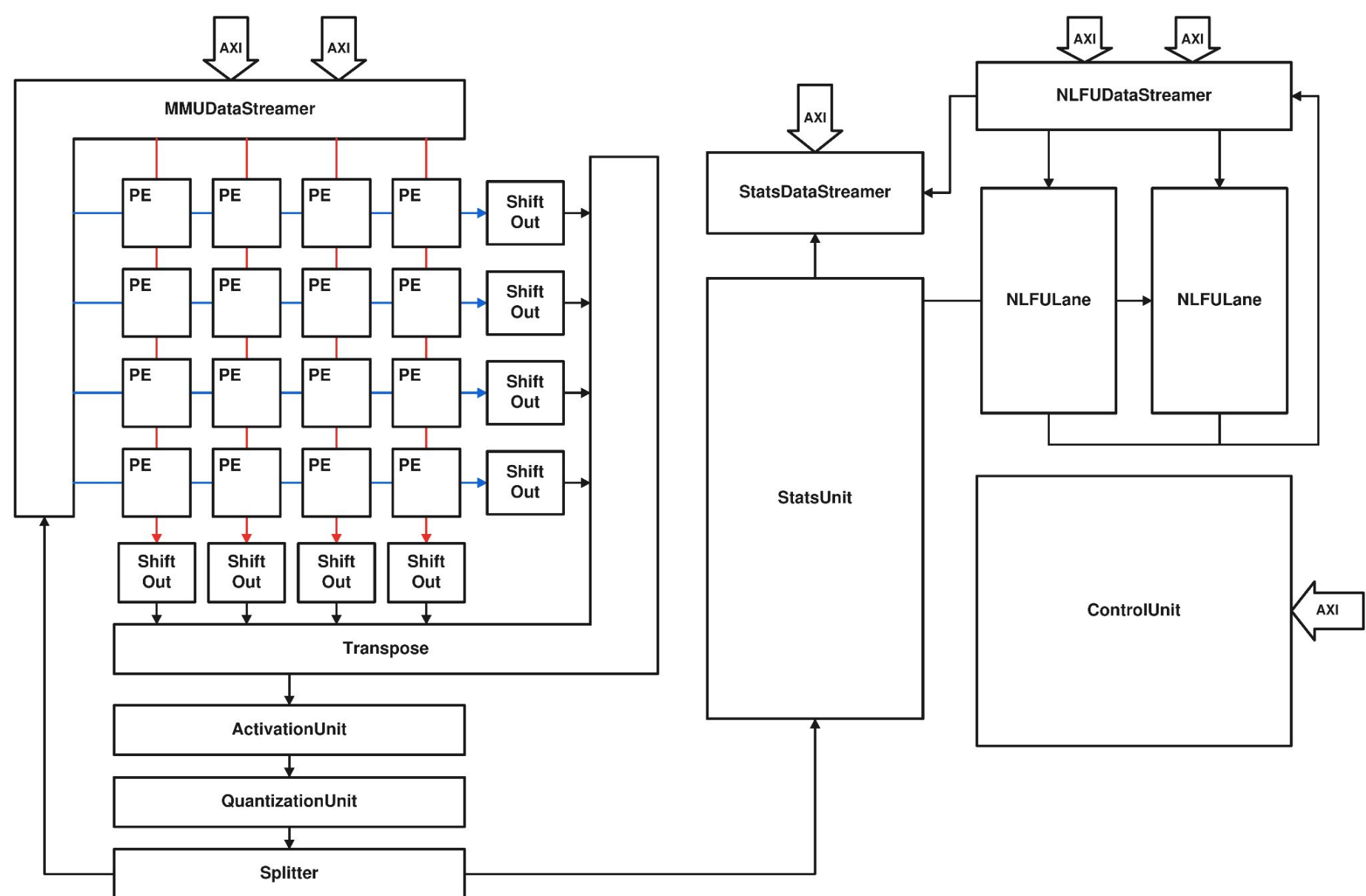
μVLA system: Hub, Spoke, and memory chiplets connected over HyperBus.

Two Memories, One Bus

- Infineon's **HyperRAM** (DRAM) and **SemperFlash** (NOR flash) both speak HyperBus-family protocols, but serve different roles: HyperRAM is a fast scratchpad for ongoing calculations; SemperFlash stores model weights, read often but almost never rewritten. This is advantageous as Flash doesn't need to refresh, so **we can read sequential data without interruption**.
- Existing controllers only support HyperRAM. SemperFlash differs enough at the **physical and protocol layers** (no write masking, a permanently-driven RWDS line, an SPI boot mode) that the controller needed a real redesign.

Conclusions & Next Steps

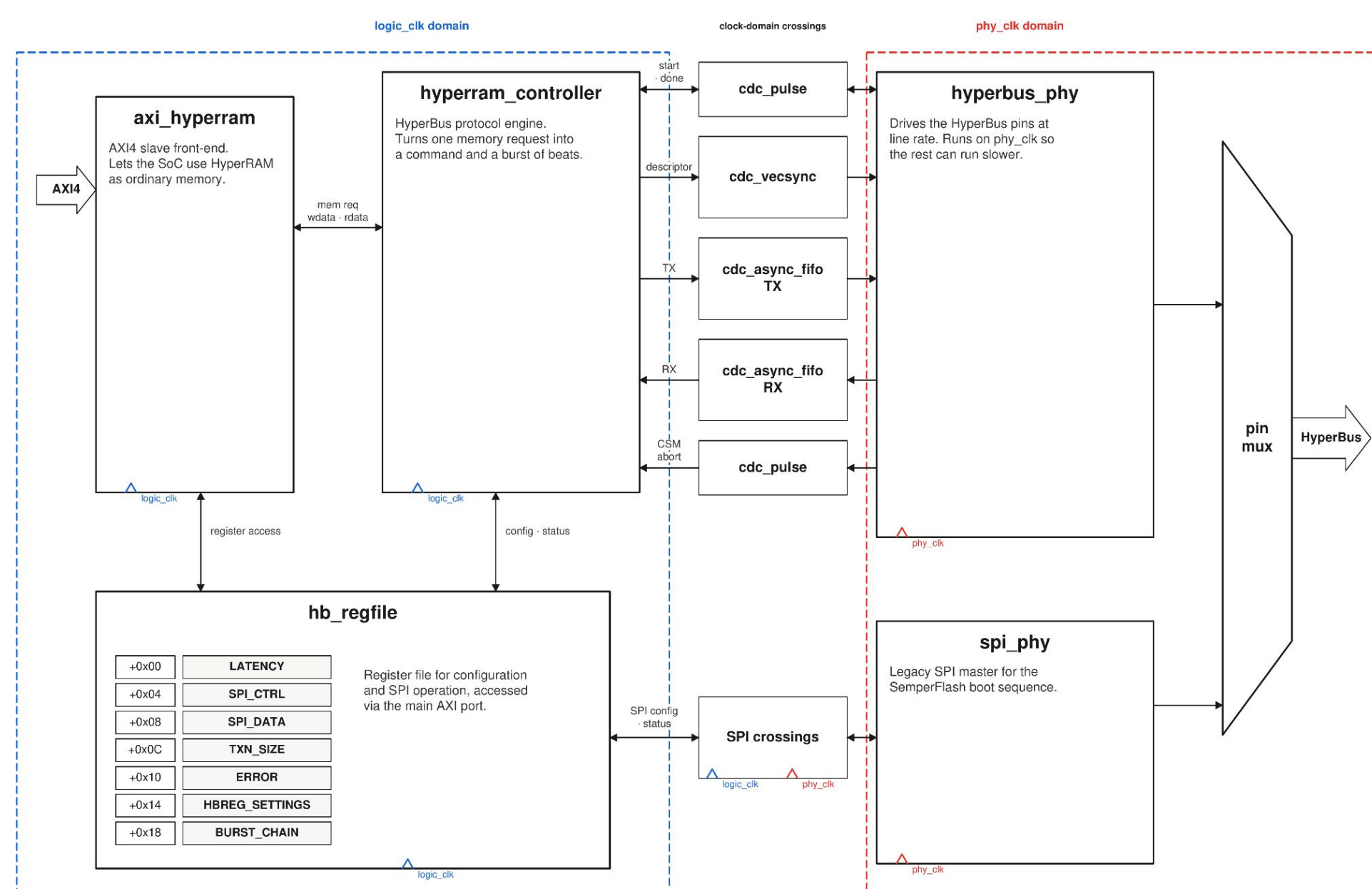
- The redesigned controller now supports **both HyperRAM and SemperFlash** on a single verified HyperBus interface, **reaching SemperFlash's full 400 MB/s theoretical bandwidth**.
- Next: **integrate the AI accelerator** (built with Catapult HLS) into the Spoke chiplet, then **compile workloads**, first for a single SoC, eventually for the full multi-chiplet system.



AI accelerator datapath — the next integration target.

Approach: A Unified HyperBus Controller

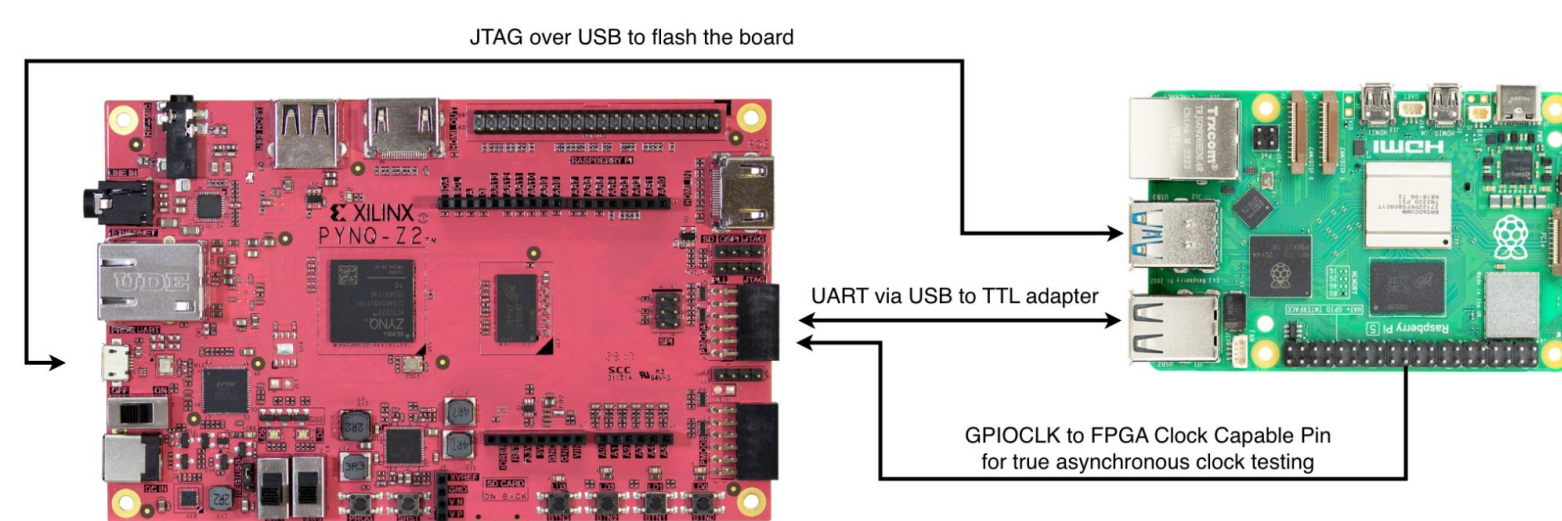
- **Extended open-source HyperRAM controllers with AI-assisted development**, then rebuilt the **integration and system-level verification by hand**. The AI-generated logic needed integration testing than block-level testbenches can't provide.
- The on-chip memory bus only needs to run near 200 MHz — plenty for HyperRAM and SemperFlash. But **chip-to-chip links reuse HyperBus at a much higher clock**, since that's the fastest way to move data between chiplets without adding pins. Running the whole controller at that higher frequency would over-stress the logic design, so the **controller splits into a logic clock domain** and a separate, faster **PHY clock domain**, bridged by a CDC block.
- Added a second PHY for **SemperFlash's SPI boot mode**, sharing I/O with the existing HyperBus PHY through pin muxing, and moved sideband control registers to memory-mapped registers accessed over AXI.



Top-level controller: logic vs. PHY clock domains, CDC bridges, register map, and dual HyperBus/SPI PHYs.

Verification

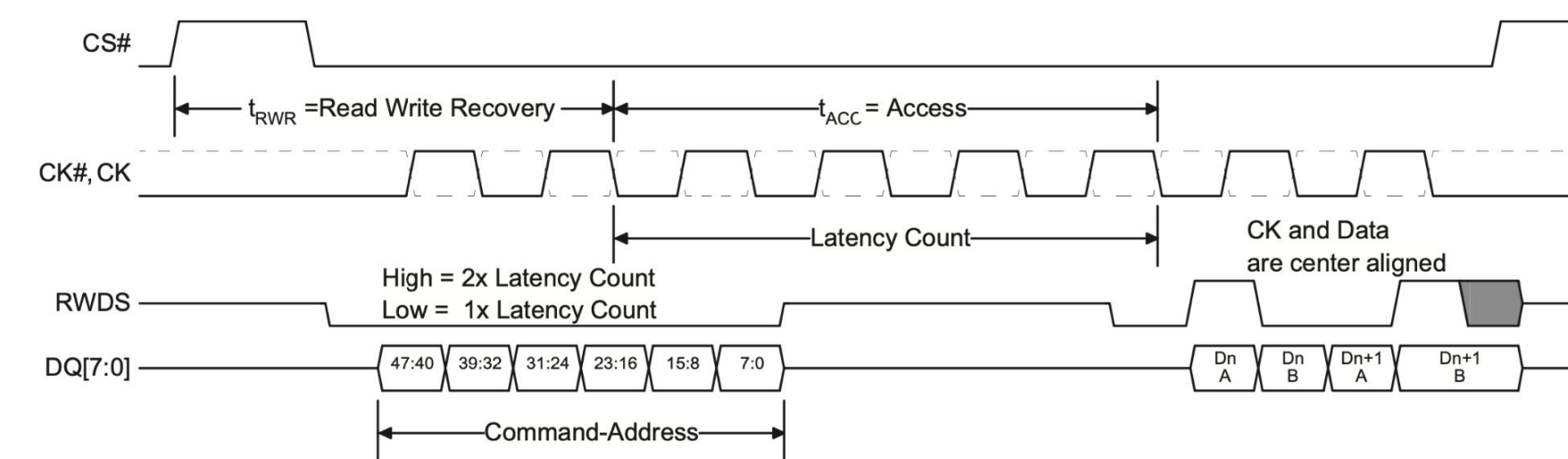
- **Fixed protocol bugs in the base HyperRAM path**: insufficient CSn-to-CK timing, and a mismatch between two different latency-count definitions.
- Verified DMA-driven 64-bit transactions and **configurable transaction-length limits**, ensuring **HyperRAM gets time to refresh** even when memory traffic is high.
- Fixed **SemperFlash-specific bugs**: non-multiple-of-4 addressing, and zero-latency writes that previously only worked for register access.
- **Stress-tested every clock-domain crossing on a PYNQ-Z2 FPGA**, using a Raspberry Pi 5 as an independent reference clock for **true asynchronous testing**, plus a full verification of functionality in VCS simulation covering latency, SPI startup, burst-chain edge cases, and error handling.



PYNQ-Z2 FPGA + Raspberry Pi 5 clock-domain-crossing test rig.

Anatomy of a HyperBus Transaction

Every transaction moves through three phases. The optimizations below work by **skipping or hiding CA and Latency**.

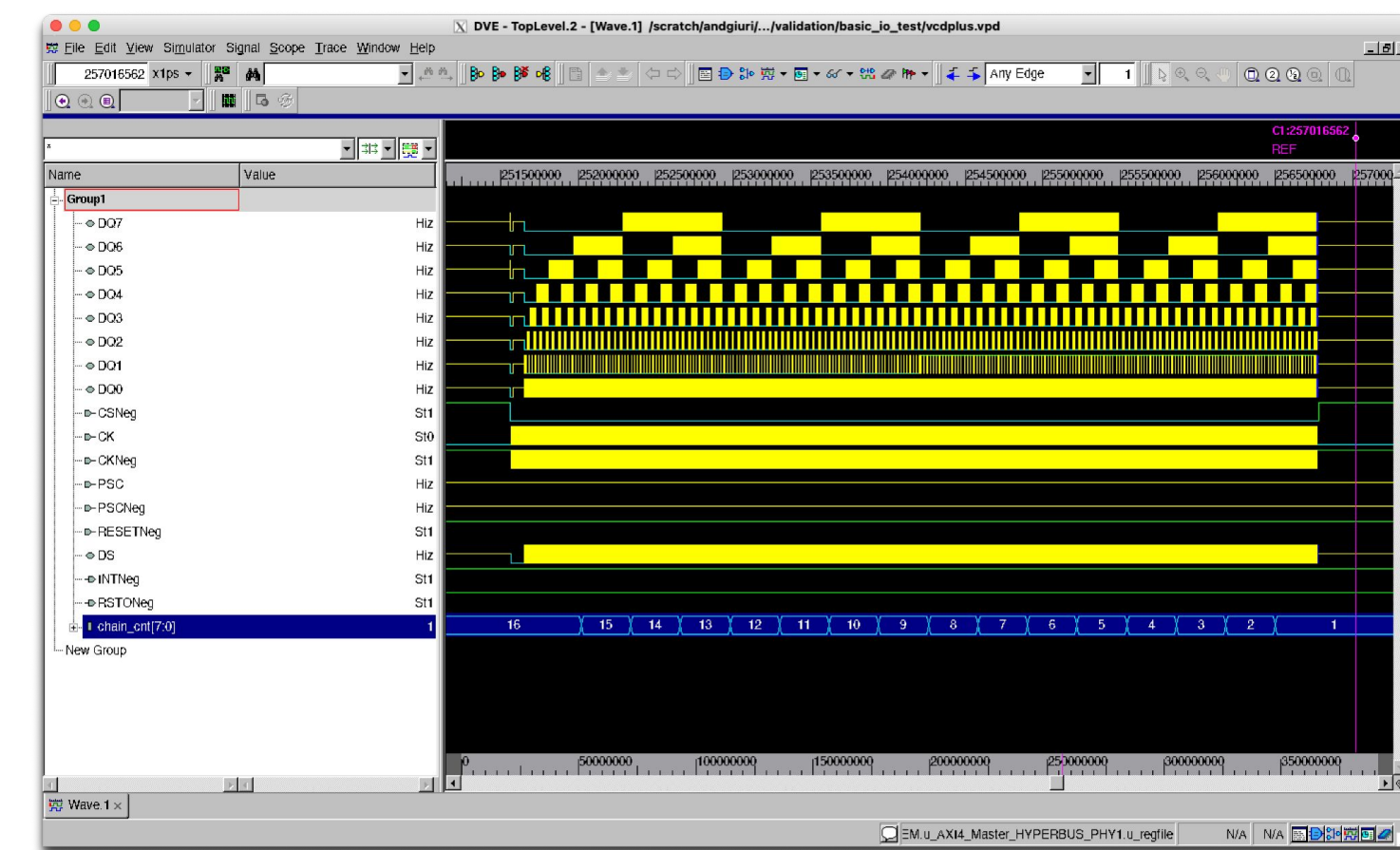


HyperBus Write Transaction

How the Optimizations Work

- **Burst chaining removes the CA and Latency** phases from every read after the first: as long as reads stay contiguous, the controller keeps CS# asserted, so only the Data phase repeats back-to-back instead of re-running the command and waiting through latency each time.
- Even with burst chaining, moving data from the PHY into the AXI bus and having the DMA line up the next request takes time. **Speculative reads use the chain length to predict the next read and start the HyperBus transaction before the request even arrives**, so the controller's own data-shuttling overhead runs in parallel with the transaction stream instead of stalling it.

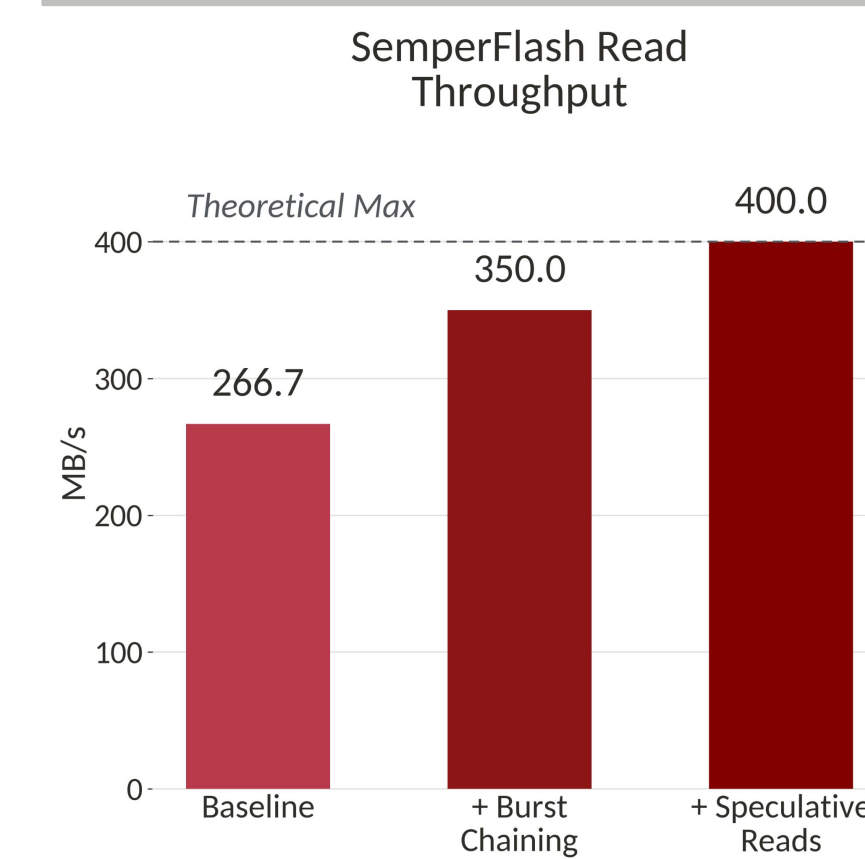
Speculative Reads in Action



DQ lines and chain counter during a 16-beat speculative read burst.

With the next read started before it's even requested, the transfer stays fully continuous, clock never stops and DQ lines never idle, so a long read reaches SemperFlash's full 400 MB/s bandwidth.

Results: Closing the Bandwidth Gap



Burst Chaining gave a great boost in performance as the CA + Latency phase of the HyperBUS transaction was the largest overhead.

Speculative Read helps close the gap fully on long reads, and given that we can directly control how the model weights will be stored in memory it will be useful in production.