Design eines Redis Cache

Uni-Projekt: Chip Design & Verilog

Philipp Hecht Luca Pinnekamp Luca Schmid

27. Februar 2026

Zusammenfassung

Implement a Redis-inspired cache that works in conjunction with the CPU. The goal is to create a fast cache for storing key-value pairs. Basic CRUD (Create, Read, Update, Delete) implementation

Inhaltsverzeichnis

1 Einleitung / Idee

Link zur custom Hardware Link zu Risc-V Tapeout

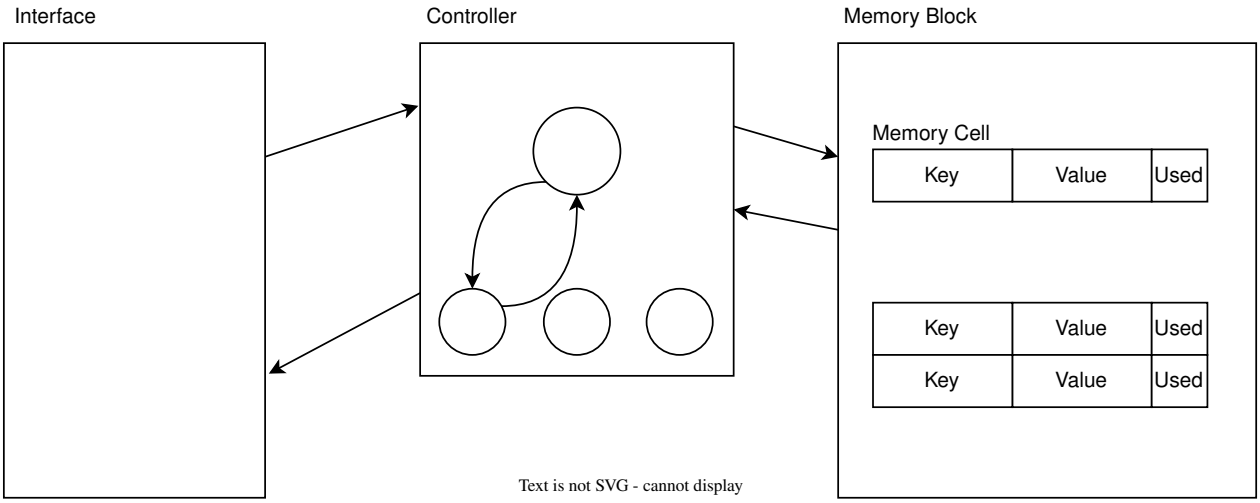
Die ursprüngliche Idee dieses Projekts ist der Entwurf und die Implementierung eines kompakten, synthetisierbaren Key-Value-Stores, inspiriert von Redis, auf RTL-Ebene (für FPGAs oder ASICs).

TODO: Ausformulieren, warum wir uns hierfür entschieden haben -> Semirealer use case; Vorstellbar was genau passieren soll -> Da redis bekannt

Dabei war das Ziel, grundlegende Speicheroperationen direkt in Hardware abzubilden, um eine hohe Performance und geringe Latenz zu erreichen. Die geplanten Kernfunktionen sind:

- **Einfügen von Schlüssel-Wert Paaren (Key-Value Insertion)**
- **Abrufen von Werten anhand von Schlüsseln (Value Retrieval)**
- **Löschen von Werten anhand von Schlüsseln (Key Deletion)**
- **Auflisten von Schlüsseln (Key Listing)**
- **Automatische Ablaufzeit (TTL - Time-to-Live)**

Die Motivation liegt darin, die Effizienz von Key-Value-Speichern durch Hardwarebeschleunigung zu untersuchen und eine Schnittstelle bereitzustellen, die ähnlich wie Software-Caches funktioniert, aber die Vorteile dedizierter Hardware nutzt.



TODO: Ich würde zunächst nur Controller und Memory Block zeigen

2 Projekt Setup

2.1 Repo Struktur

Zunächst befassten wir uns mit dem Aufbau einer generellen Code Struktur sowie dem Aufsetzen von #Pipelines. Hierfür wurde zunächst ein *Fork* des vom Dozenten bereit gestelltem GitHub Repostory aufgesetzt. Der *Fork* wurde verwendet um eine generelle Vorgabe für die Code Struktur zu erlangen. Darauf aufbauend wurden folgende Verzeichnisse angelegt:

- .github/Workflows
- docs
- src TODO: Ausweiten auf die einzelnen Module

Zusätzlich wurden eigens geschriebene Dockerfiles sowie Makefile Targets erstellt, welche uns ermöglichen, komfortabler während der Projektphase zu arbeiten. Innerhalb der Dockerfile werden benötigte Bibliotheken (bspw. Verilator) installiert. Die Makefile ermöglicht es uns, rekursiv die einzelnen Sub-Module des Projektes () zu bauen als auch zu testen.

2.2 Pipelines

Als ersten wichtigen Punkt für das Zusammenarbeiten während der Projektphase wurde das Aufsetzen von Pipelines angegangen. Hierbei wurden zwei GitHub Workflows implementiert, welche unabhängig voneinander eine Frontend / Backendpipeline triggern. Aufgabe der Frontend Pipeline ist das ausführen sämtlicher Tests für die Submodule als E2E Tests der gesamten Hardware.

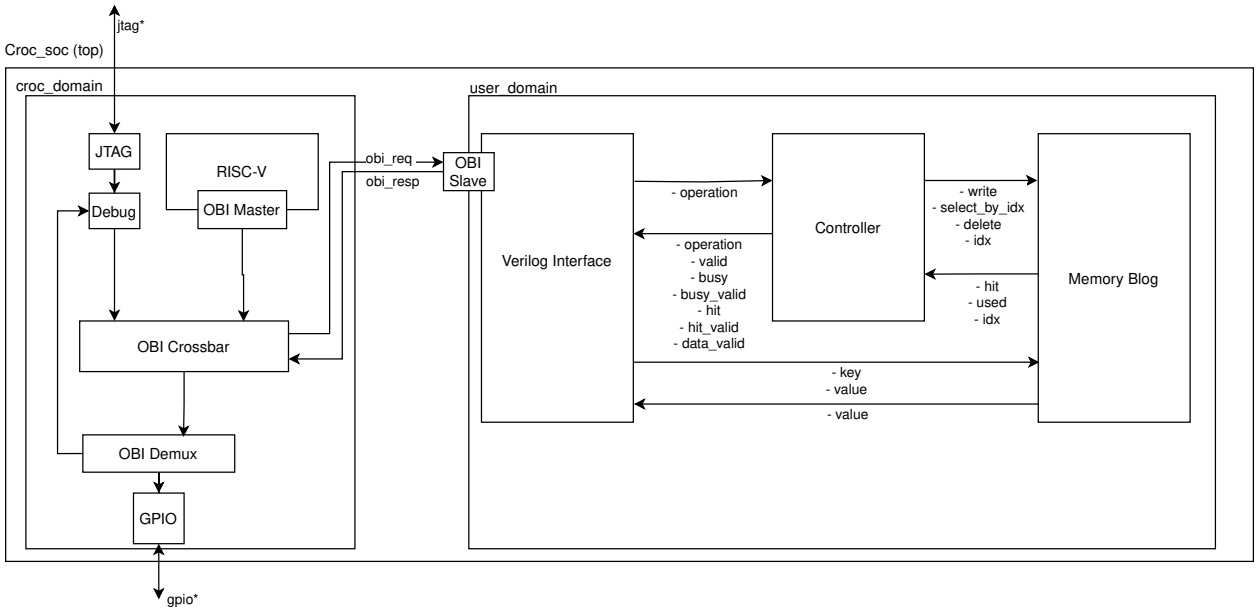
Die Backend Pipeline wird beim mergen auf den *Main*-Branch aufgerufen und führt die in der Einleitung gezeigten Librelane Pipeline aus.

TODO: weiteres Repository für tapeout darstellen; TODO: Detailierter auf Pipeline eingehen (act zum lokalen testen anmerken?)

TODO: ausblick, dass backendpipeline nie ausgeführt wurde, da keine Zeit, zu komplex und eigentlich falsches Repo?

3 Architektur

3.1 Alles richtung Architektur (?)



TODO: Darstellung des Prozesses zur Architektur TODO: Erklärung der einzelnen Module

TODO: Verlinkung auf Implementierungs.md

3.2 Statemachine (Luca S)

Ursprünglich wurde GET, UPSERT und DELETE mit mehreren States designed. Bei der Implementation ist aufgekommen, dass alles mit einem Sate geht. Die Implementation für DELETE

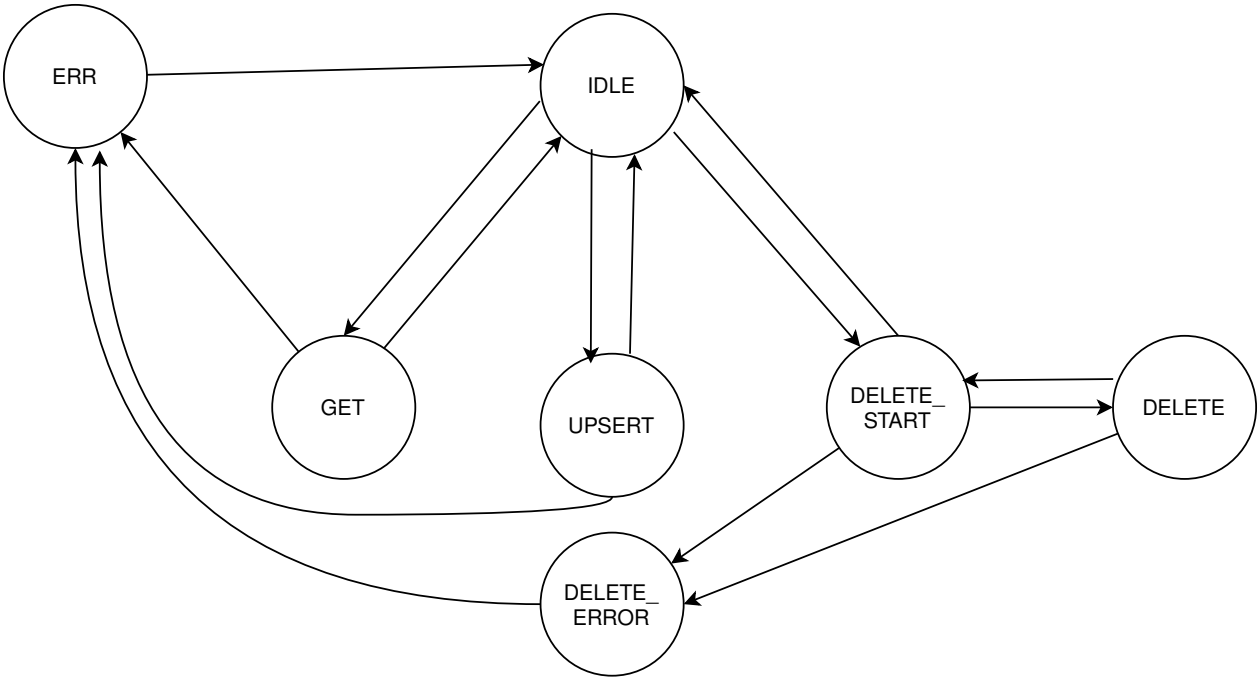


Abbildung 1: Statemachine

wurde aus Zeitrgründen nicht refactored. TODO: (Änderung der Implementation im Ausblick beschreiben oder zumindest in eine Liste eintragen)

TODO: erklären, warum wir sub states wollten TODO: erklären, Optimierung der States (Insert zunächst drei Zyklen und dann runter auf zwei)

```
Beispiel Code wie bei uns die Statemachine in die einzelnen Operationen unterteilt wurde.
cmd.done, cmd.error übergabe
en und enter
```

3.3 Taktzyklus Beispiele (Luca S)

TODO: darstellen der initialen Planung wie wir nur zw. controller und memory block arbeiten wollten TODO: Takt Diagramm erstellen, welches genau auf die einzelnen Phasen eingeht

Abbildung 2: taktzyklus

4 Implementationen

4.1 Memory (Philipp)

4.2 Controller

4.2.1 Main Controller (Luca S Luca P)

4.2.2 GET (Luca P)

4.2.3 UPSERT (Luca S)

4.2.4 DELETE (Philipp)

4.3 Obi interface

TODO: Verlinkung auf obi.md für Protokoll beschreibung

5 CROC (Luca P)

5.1 CROC Architektur

croc_soc
croc_domain
user_domain
alles andere

5.2 Theorie

5.3 Implementation bei uns im Projekt

TODO: Einbindung TODO: anpassen der Pckgs. -> Interface implementierung TODO: Conversion von bender auf librelane TODO: Pulp IPs für OBI standard

6 OBI

6.1 OBI 1 Versuch (Philipp)

6.2 OBI 2 Versuch (Luca P)

6.3 OBI Protokoll

6.4 Register

Das OBI Protokoll wird genutzt um Daten in Register zu schreiben und daraus auszulesen. Standardmäßig sind folgende Register vorhanden:

	Value	Key	Ctrl
Größe	64 (8 Bytes)	32 (4 Bytes)	32 (4 Bytes)
Offset	0	8	12

Die Größe der Register ist in der cache_cfg_pkg. Die Offset Werte leiten sich aus den den Größen

der Key und Value Register ab. In den OBI Nachrichten wird das Offset (addr) mitgegeben. Anhand des Offsets können Daten in das richtige Register geschrieben werden.

6.5 OBI Request

Die OBI Request wird von dem Master an den Slave (Interface) verserndet. Die 72 Bit lange Nachricht setzt sich aus dem Adress-Channel und dem Kontrollsignal zusammen.

Feld in obi_req_t	Bit-Breite	Beschreibung
addr	32	Register Speicheradresse: Der Wert entspricht dem Offset um die übermittelten Daten in dem richtige Register zuzuweisen
we	1	Write Enable: 1 bedeutet Schreiben, 0 bedeutet Lesen.
be	4	Byte Enable: Gibt an, welche Bytes der 32-Bit-Daten (wdata) gültig sind. Für ein volles 32-Bit-Wort ist das 1111
wdata	32	Write Data: Die Daten, die in den Speicher/das Register geschrieben werden sollen.
aid	1	Address ID: Eine ID für die Transaktion
a	1	Optional: Ein optionales Signal des OBI-Standards (in der Minimal-Konfiguration 1 Bit groß).
req	1	Request: Das Handshake-Signal. Wenn 1, bittet der Master um eine Transaktion.

6.6 OBI Response

Die OBI Response wird von dem Slave (Interface) an den Master versendet. Die 37 Bit lange Nachricht setzt sich aus dem R-Channel, dem Grant un dem Valid Signalen zusammen.

Feld in obi_rsp_t	Bit-Breite	Beschreibung
rdata	32	Read Data: Die Daten, die vom Interface gelesen wurden
rid	1	Response ID: Spiegelt die aid aus dem Request wider, um Antworten zuzuordnen
err	1	Error: Wird 1, falls beim Zugriff ein Fehler aufgetreten ist (z. B. falsche Adresse).

Feld in obi_rsp_t	Bit-Breite	Beschreibung
r	1	Optional: Ein optionales Signal für die Response (in der Minimal-Konfiguration 1 Bit).
gnt	1	Grant: Handshake-Signal. Der Slave setzt dieses Bit auf 1, um zu signalisieren: “Ich habe deinen Request (req=1) akzeptiert und verarbeite ihn”
rvalid	1	Response Valid: Wird 1, wenn die zurückgegebenen Daten in r.rdata gültig sind.

TODO: Versuch der eigenen Implementierung, sowie deren Scheitern und umschwung auf bereits bestehendes + Abwandlung

TODO: Darstellung in Implementierung

7 FPGA (Luca P)

TODO: Luca

8 C Lib (Luca P)

TODO: Einbindung in Croc

9 Backend (Philipp)

10 Learnings (alle)

11 Ausblick / Zusätzliche Funktionalitäten (Alle)

- DELETE Implementation refactoren. Kann auf 1 State gekürzt werden
- LIST Operation

12 Vivado Setup - Mac Anleitung (Luca P) Optional!!!!