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

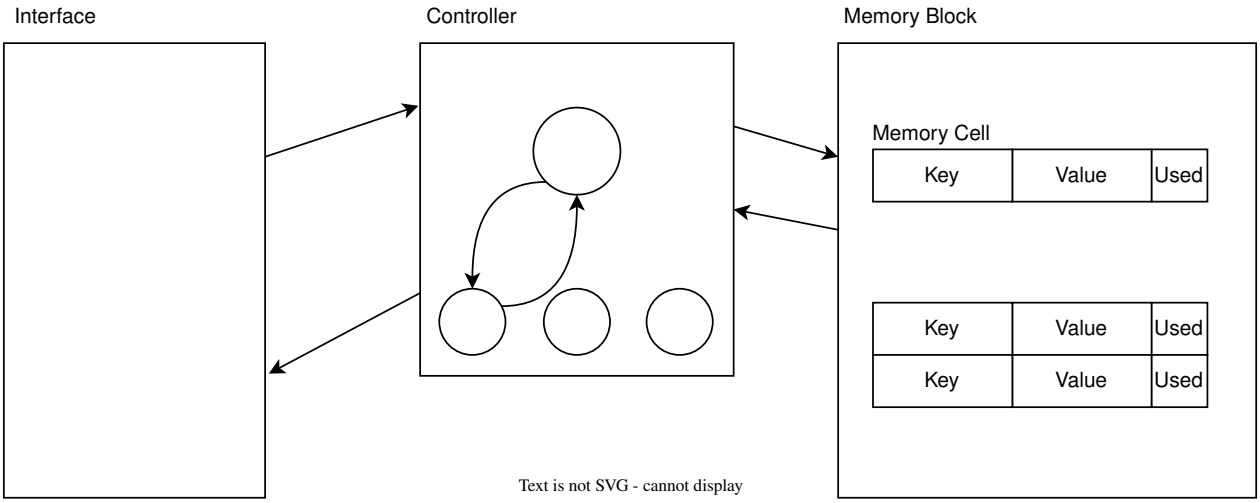
Die ursprüngliche Idee dieses Projekts ist der Entwurf und die Implementierung eines kompakten, synthetisierbaren Key-Value-Stores, inspiriert von Redis, auf RTL-Ebene (für FPGAs oder ASICs).

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 (?)

Die Architektur ist in die CROC Architektur eingebettet. Die CROC_DOMAIN wird im Kapitel CROC beschrieben. Die USER_DOMAIN ist modular aufgebaut und in drei Hauptkomponenten unterteilt:

1. **OBI Interface (`obi_interface`)**: Dieses Modul fungiert als Slave am OBI-Bus. Es nimmt Anfragen entgegen und verwaltet die internen Register für Key, Value und Control-Signale. Es entkoppelt das Bus-Protokoll von der internen Logik.
2. **Controller (`controller`)**: Die zentrale Steuereinheit (Orchestrator). Sie liest die Control-Register, interpretiert die Befehle (GET, UPSERT, DELETE) und steuert die Schreib-/Lese-Signale des Speichers. Die Ausführung komplexer Abläufe delegiert der Controller an spezialisierte Sub-FSMs (siehe Implementation).
3. **Memory Block (`memory_block`)**: Enthält das eigentliche Speicher-Array. Die Daten (Key/Value) liegen direkt aus den Interface-Registern am Speicher an.

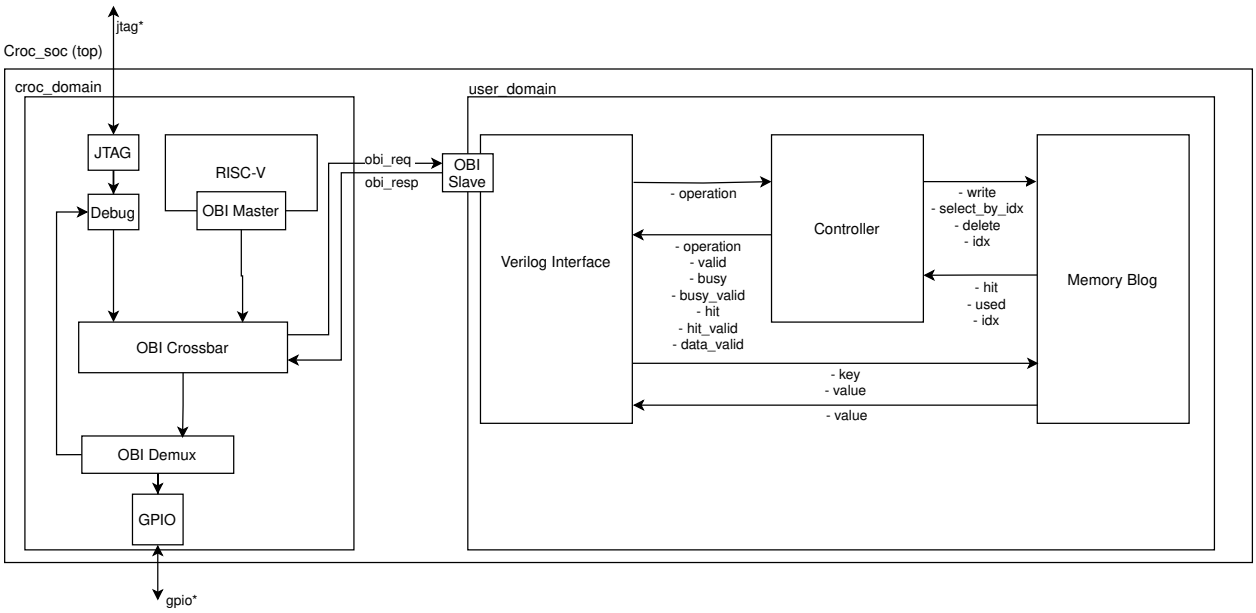


Abbildung 1: Architektur

3.2 Statemachine (Luca S)

Zu Beginn des Projekts wurde die State Machine mit einem stark sequenziellen Ansatz entworfen, ähnlich einem Software-Ablaufplan. Dabei wurden komplexe Operationen wie UPSERT in viele nacheinander ablaufende Sub-States unterteilt.

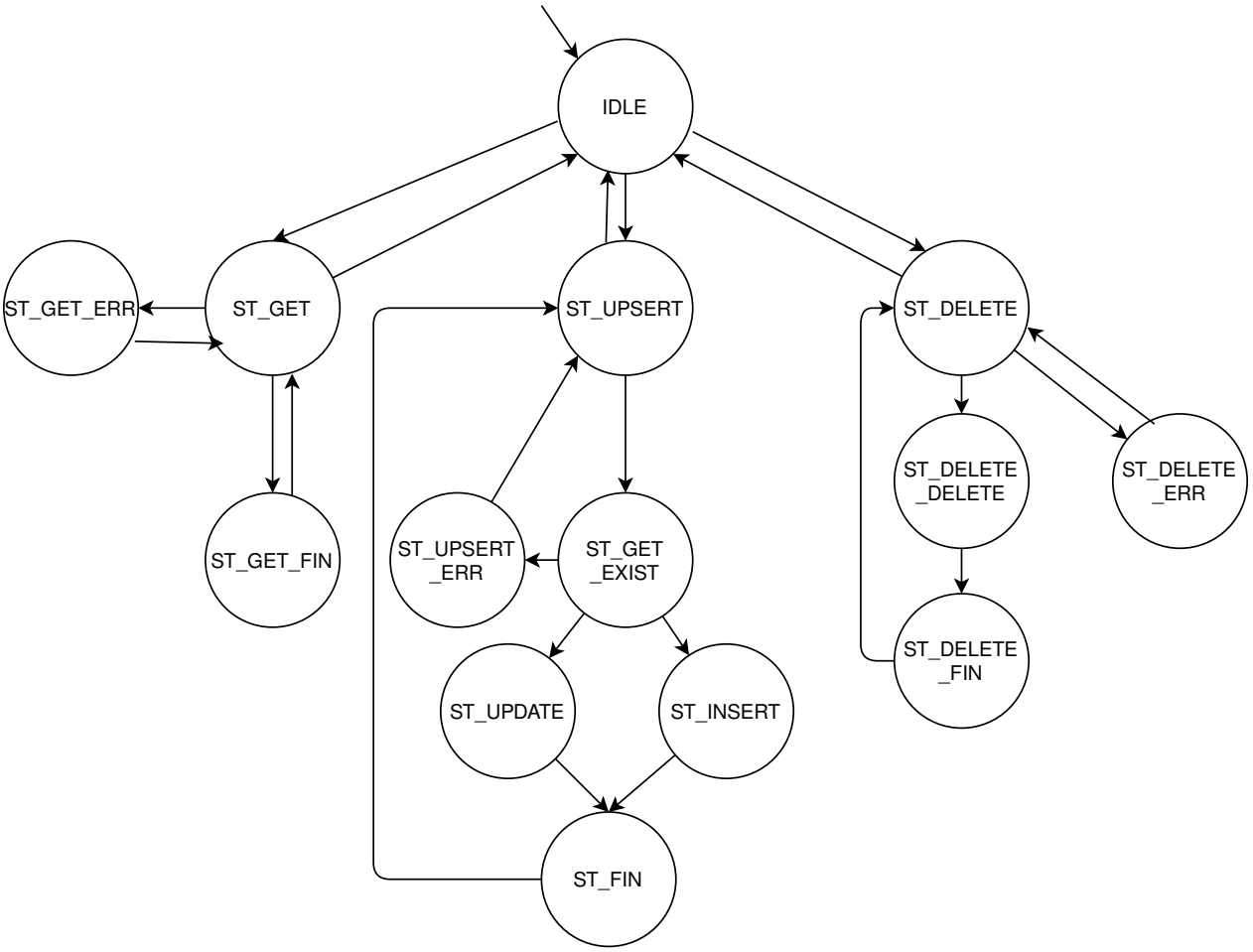


Abbildung 2: Initiale Statemachine

Dieser Ansatz hätte Lese- und Schreiboperationen künstlich über mehrere Taktzyklen gestreckt, da in jedem Taktzyklus nur eine einzige Bedingung evaluiert wurde.

Während der Implementierung zeigte sich, dass durch die Hardware-Parallelität Signale kontinuierlich anliegen und die Bedingungen gleichzeitig (kombinatorisch) geprüft werden können. Somit konnten wir die Statemachine deutlich verkleinern und auf nur einen Zyklus pro Operation verkürzen.

Hinweis: Aus Zeitgründen wurde die DELETE Operation noch nicht auf das optimierte Konzept umgestellt und entspricht noch dem initialen, sequenziellen Ansatz.

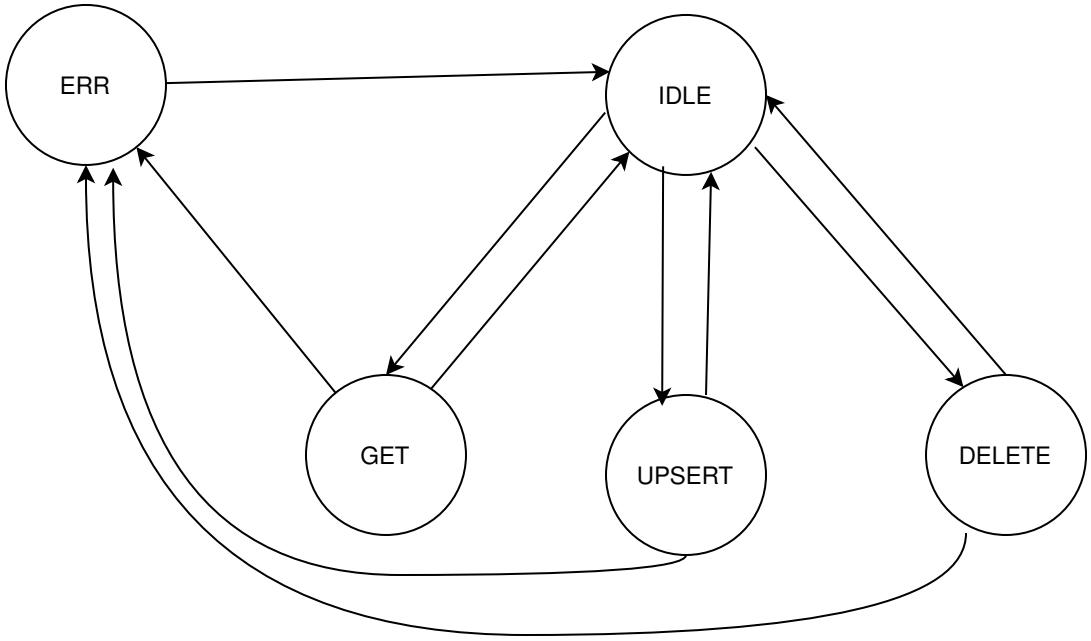


Abbildung 3: Statemachine

3.3 Taktzyklus Beispiele

Basierend auf der ursprünglichen Statemachine haben wir zur eeffizienteren Verarbeitung geplant, im Controller auf positive und im Memory Block auf negative Taktflanken zu warten. Dies sollte ermöglichen, Lese- und Schreiboperationen innerhalb einer Taktperiode abzuschließen. Folgendes Diagramm zeigt die daraus resultierenden Taktflanken.

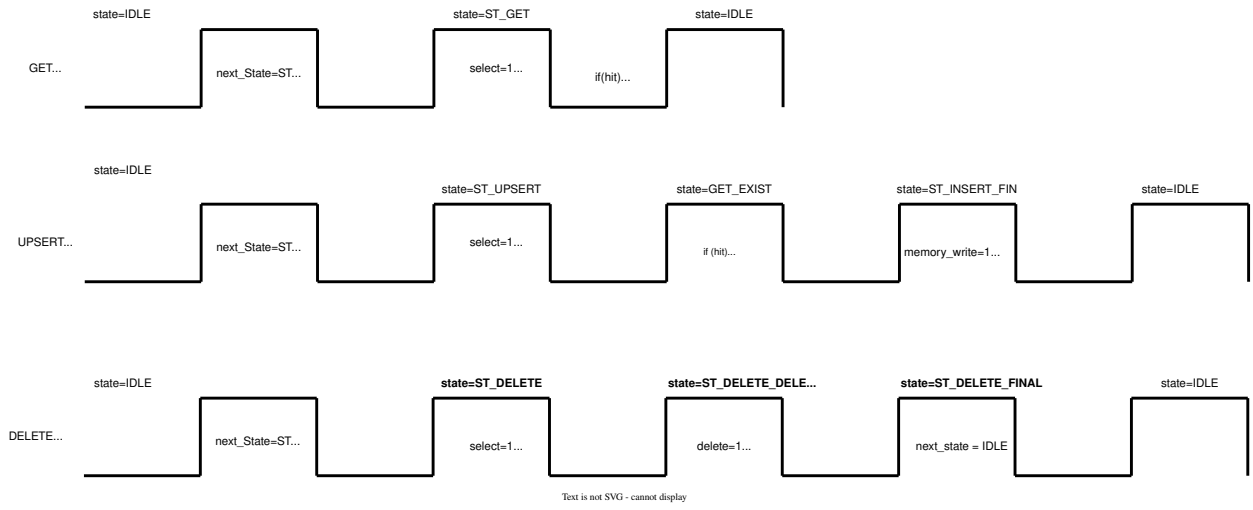


Abbildung 4: taktzyklus

Um jedoch potentielle Timing-Probleme zu vermeiden wurden der Memory Block auf die positive Taktflanke umgestellt. Mithilde der Python-Tests (Cocotb) konnten die tatsächlichen Taktzyklen verifiziert werden. Siehe Kapitel Tests.md

Abbildung 5: GET

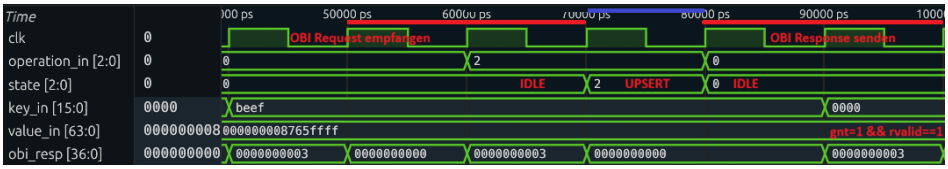


Abbildung 6: UPSERT (empty memory)

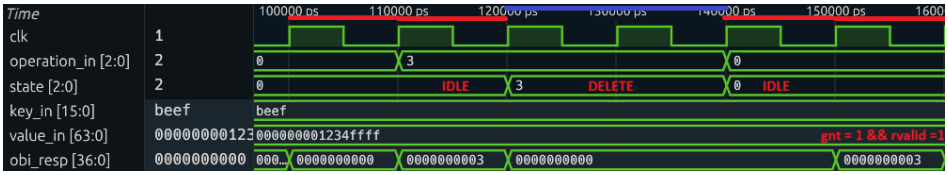


Abbildung 7: DELETE (key exists)

4 Implementationen

4.1 Memory (Philipp)

4.2 Controller

4.2.1 Main Controller (Luca S, Luca P)

Der Main Controller (controller.sv) dient als Orchestrator der Operationen. Seine Hauptaufgabe besteht darin, eingehende Operationen von dem Interface entgegenzunehmen und die Ausführung an spezialisierte Sub-Controller (GET, UPSERT, DELETE) zu delegieren.

Die Architektur ist hierarchisch aufgebaut. Der Main Controller implementiert eine übergeordnete State Machine, die im `IDLE`-Zustand auf Anfragen wartet. Sobald eine valide Operation erkannt wird, wechselt der Controller in den entsprechenden Zustand und aktiviert das zuständige Sub-Modul.

Schnittstelle zu Sub-Controllern

Um die Komplexität zu kapseln, verfügen alle Sub-Controller über ein einheitliches Interface-Konzept zur Kommunikation mit dem Main Controller:

- **`en` (Enable):** Ein Signal vom Main Controller an den Sub-Controller, um dessen FSM oder Logik zu starten.
- **`cmd` (Command/Data):** Status Signale der Sub-Controller. (Error/Done)
- **`enter` (Enter):** Signalisiert den Eintritt in den Zustand des Sub-Controllers. Dient zur Initialisierung der Sub-FSM.

Dies ermöglicht es dem Main Controller, generisch auf das Ende einer Operation zu warten, ohne die internen Details der Operation kennen zu müssen.

```
// Pseudo-Code Beispiel für die State-Wechsel vom Main Controller zu den Sub-Controllern
always_comb begin
    // Default Zuweisungen
    next_state = state;

    case (state)
        IDLE: begin
            busy = 1'b1;

            case (operation_in)
                    READ: begin
                        next_state = ST_GET;
                    end
                    // ... Weitere Operationen ...
            endcase
        end
        ST_GET: begin
            // ...                                  Werte an Sub-Controller übergeben
            if (get_error) next_state = ST_ERR; // Error handling
            else if(get_done) begin             // Operation erfolgreich
                next_state = IDLE;
                busy = 1'b0;
            end
        end
        // ... Weitere Zustände ...
    endcase
end
```

4.2.2 GET (Luca P)

4.2.3 UPSERT (Luca S)

Der UPSERT-Controller (“Update or Insert”) ist für das Schreiben von Daten in den Cache verantwortlich. Er wurde so implementiert, dass er unabhängig vom Basistemplate agiert und die spezifische Logik für das Hinzufügen oder Aktualisieren von Key-Value-Paaren kapselt.

Die Key-Value Werte liegen direkt vom OBI-Interface am Memory-Block an. (Siehe Kapitel Architektur) Somit steuert die Implementierung nur noch die Signale zur Verarbeitung dieser Werte im Memory-Block.

Wenn der empfangene Key-Wert bereits im Memory vorhanden ist, dann gibt der Memory-Block dem UPSERT-Controller ein positives “hit” signal und dessen “index” zurück. Wenn der Key-Wert nicht vorhanden ist liegt ein negatives “hit” signal an. Außerdem übergibt der Memory Block die aktuell benutzen Indizes “used”.

Anhand der Werte hit, idx_out und used kann in nur einem Zyklus entschieden werden, ob oder an welchem Index der anliegende Wert gespeichert werden soll:

```
// Pseudo-Code Beispiel UPSERT
always_comb begin
    if (hit) begin
        // key existiert
        // Übergebenen Index an Memory Block übergeben

    end else if (!hit && !(&used)) begin
        // key existiert nicht, aber noch Platz im Memory
        // korrekten key finden und übergeben
        for (int j = 0; j < NUM_ENTRIES; j++) begin
            if (!used[j] && (idx_out == 0)) begin
                idx_out[j] = 1'b1;
            end
        end

    end else begin
        // key existiert nicht und kein Platz im Memory
        // Error an Main Controller übergeben
    end
```

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).
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 Tests

10.1 Allgemeines Testkonzept

Um die korrekte Funktionalität der Module sicherzustellen, haben wir Frontend-Tests mittels **Cocotb** (Coroutine Co-simulation Testbench) durchgeführt. Die Tests gliedern sich in zwei Ebenen:

1. **Unit-Tests:** Überprüfung einzelner Module (z.B. `memory_block`, `dynamic_register_array`)
2. **End-to-End Tests:** Überprüfung des Gesamtsystems (`redis_cache`) über das OBI-Interface

Als Simulatoren kommen **Verilator** und **Icarus Verilog** zum Einsatz. Bei den Unit-Tests auf Modul-Ebene ohne komplexe SystemVerilog-Structs konnte Icarus Verilog verwndet werden. Für den End-to-End Test auf der Top-Ebene wurde aufgrund der integrierten OBI-Packages und Structs der Simulator Verilator benötigt.

10.2 Unit-Tests

Bevor das Gesamtsystem getestet wurde, wurden die einzelnen Module wie Controller, Interface und Speicherkomponenten einzeln getestet. Beispielsweise prüft der Test `test_memory_block.py` die direkte Ansteuerung des Speichers ohne den Overhead des OBI-Protokolls.

Hierbei werden Szenarien abgedeckt wie: - Schreiben und Lesen von Key-Value Paaren. - Verhalten bei vollem Speicher (`used_entries`). - Löschen von Einträgen und Überprüfung der Persistenz nicht gelöschter Daten. - Überschreiben bestehender Einträge.

10.3 End-to-End Test (`test_redis_cache.py`)

Der zentrale Bestandteil der Verifikation ist der Integrationstest `test_redis_cache.py`. Dieser Test instanziiert das Top-Level-Modul `redis_cache`, in dem alle Sub-Komponenten (`obi_interface`, `controller`, `memory_block`) miteinander verbunden sind.

10.3.1 Funktionsweise

Dieser Test simuliert die Sicht eines externen Masters (z.B. einer CPU). Es werden ausschließlich Daten über das OBI-Interface verschickt und empfangen.

1. **Daten schreiben:** Key und Value werden an die entsprechenden Register-Adressen des Interfaces gesendet.
2. **Kommando senden:** Der Opcode (GET, UPSERT, DELETE) wird in das Control-Register geschrieben.
3. **Warten:** Die Testbench wartet, bis der Controller die Operation verarbeitet hat und das gnt Signal zurück schickt.
4. **Verifikation:** Das Ergebnis (z.B. gelesene Daten oder Status-Bits) wird überprüft.

10.3.2 Code-Beispiel: Ausführen einer Operation

Die Hilfsfunktion `execute_cache_operation` in der Testbench mapped die Daten in die entsprechenden Register-Adressen, bevor sie versendet werden:

```
async def execute_cache_operation(dut, tester, operation, key, value=0):

    # 1. Value Register schreiben (nur bei UPSERT nötig)
    if operation == 'UPSERT':
        await obi_write(dut, addr=0, wdata=value)

    # 2. Key Register schreiben (Adresse 8)
    await obi_write(dut, addr=8, wdata=key)
```

```
    # 3. Kommando im Control-Register absetzen (Adresse 12)
    await obi_write(dut, addr=12, wdata=(op_code << 1), be=1)

    # 4. Warten bis Controller fertig ist (Polling auf State 0/IDLE)
    while int(tester.u_ctrl.state.value) != 0:
    await tester.wait_cycles(1)
```

Die Hilfsfunktion `obi_write` erstellt die OBI Nachricht und führt den Handshake durch.

```
async def obi_write(dut, addr, wdata, be=0xF):
    dut.obi_req_i.value = pack_obi_req(addr=addr, we=1, be=be, wdata=wdata, req=1)

    # Warten auf das Grant-Signal (Handshake)
    while True:
        await RisingEdge(dut.clk)
        resp_val = int(dut.obi_resp_o.value)
        gnt_bit = (resp_val >> 1) & 1
        if gnt_bit == 1:
            break

    # Request wieder auf 0 ziehen
    dut.obi_req_i.value = pack_obi_req()
    await RisingEdge(dut.clk)
```

10.3.3 Analyse

Die während der Tests aufgezeichneten Signale werden aufgezeichnet und können im Nachgang analysiert werden. Die folgende Grafik zeigt die aufgezeichneten Signale eines `UPSERT` in einen leeren Cache. In der Zeile `operation_in` ist zu sehen, welche Operation vom Controller empfangen wird. Im darauffolgenden Takt wechselt der Controller in den entsprechenden Zustand. Nach dem einfügen des Keys in den Speicher wechselt der Controller zurück in den IDLE Status. Gleichzeitig sendet das OBI-Interface die Response. Der Wert `3` der OBI-Response bedeutet, dass das GNT und rvalid Bit gesetzt sind (Siehe Kapitel OBI) und die Operation erfolgreich durchgeführt wurde.

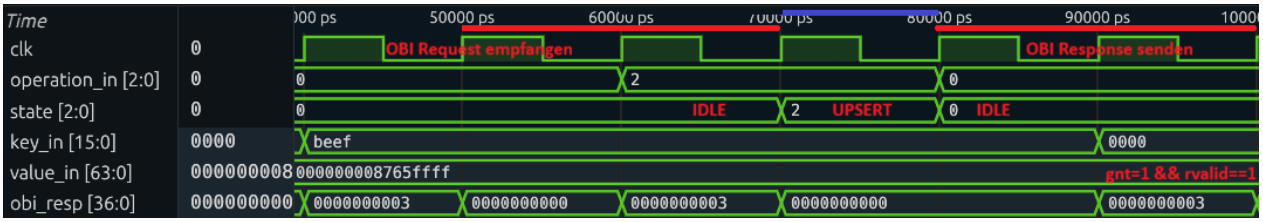


Abbildung 8: UPSERT (empty memory)

11 Learnings (alle)

12 Ausblick / Zusätzliche Funktionalitäten (Alle)

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

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