Design eines Redis Cache

Uni-Projekt: Chip Design & Verilog

Philipp Hecht Luca Pinnekamp Luca Schmid

27. Februar 2026

Zusammenfassung

Dieses Dokument beinhaltet unsere Projektdokumenation zur Vorlesung Hardware/Software-Codesign. Das Dokument beschreibt die Implementation und Funktionalität des während des Blockseminars designten Redis-inspirierten Caches.

Inhaltsverzeichnis

1 Einleitung / Idee

Ausarbeitung gemeinsam

Link zu Custom Hardware

Link zu Risc-V (Croc) Tapeout

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

Grund für diese Entscheidung war

1) die leichte Erweiterbarkeit der Implementierung: Ausgehend von einfachen Operationen konnten wir diese schrittweise erweitern.
2) Einsteigerfreundlich: Dadurch, dass keiner von uns vorher wesentliche Erfahrung mit Hardware Designs hatte, wollten wir ein möglichst leicht zu verstehendes Projekt umsetzen.
3) Semirealer Use Case: Im Gegenzug zu anderen Projekten hatten wir die Idee etwas umzusetzen, was so ggf. in der Praxis vorkommen könnte.
4) Leicht verständlich: Innerhalb unserer Gruppe war die Idee leicht verständlich, sodass alle mit einem gleichen / ähnlichem Verständnis starteten.

Grundlegendes Ziel war 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 lag 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.

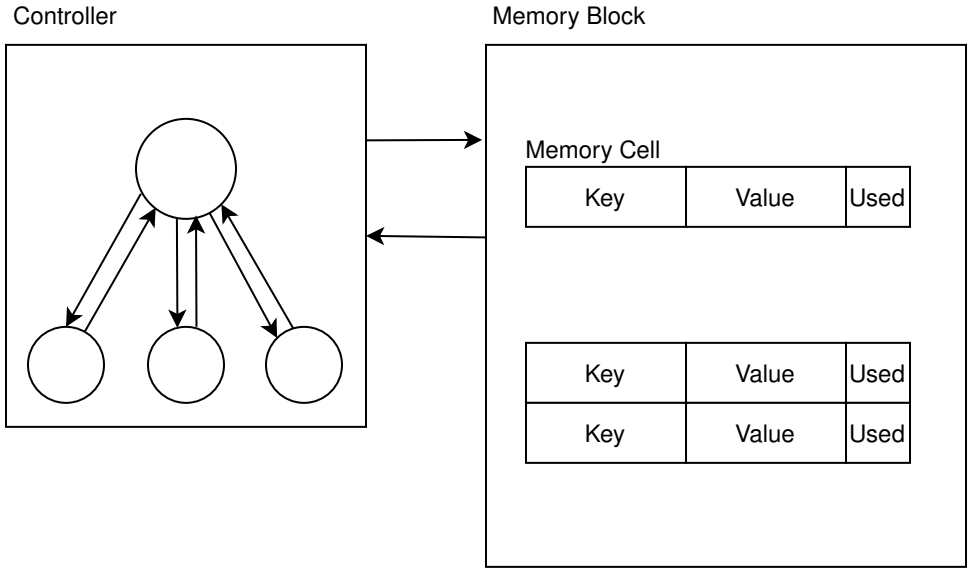

Abbildung 1: Speicherblöcke

2 Projekt Setup

Ausarbeitung gemeinsam

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
- riscv
- src
  - controller
  - interface
  - memory
  - redis_cache

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.

Näheres hierzu wird im Kapitel Pipelines beschrieben.

3 Architektur

Nachfolgendes Diagramm gibt einen Überblick über die Gesamtarchitektur unseres Redis Caches und zeigt die Verbindungen zwischen den einzelnen Modulen, auf die in den folgenden Abschnitten näher eingegangen wird.

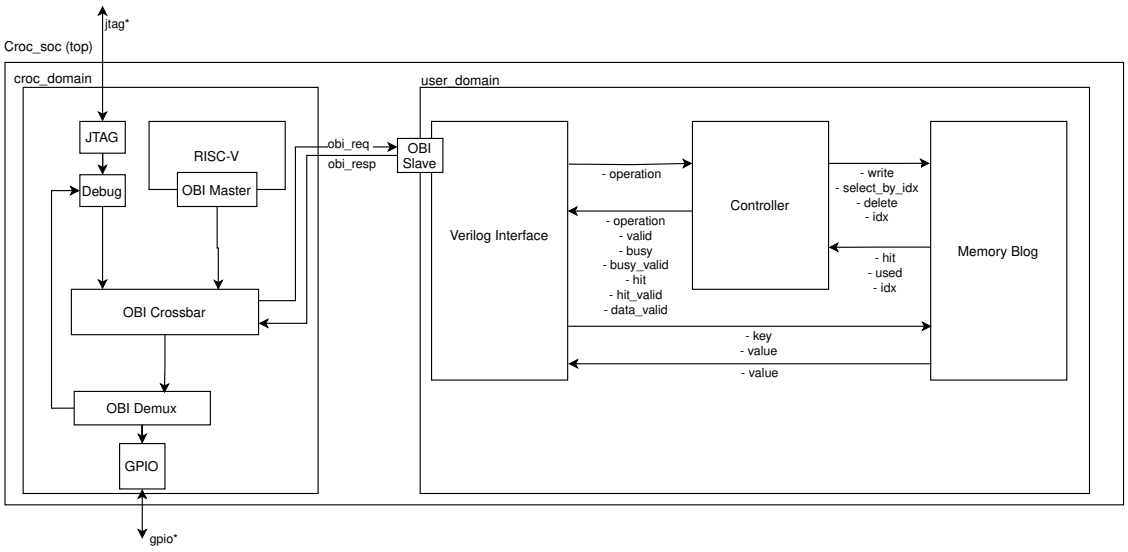

Abbildung 2: Architektur

Dabei ist unsere Cache Hardware in die CROC Architektur eingebettet. Die CROC_DOMAIN wird im Kapitel CROC beschrieben. Innerhalb der USER_DOMAIN befindet sich der von uns implementierte Redis-Like Cache, welcher über das OBI Interface mit der CROC_DOMAIN kommuniziert. Dabei besteht der Redis Cache aus den folgenden drei Hauptmodulen:

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 und entkoppelt somit das OBI 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.

3.1 Statemachine

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.

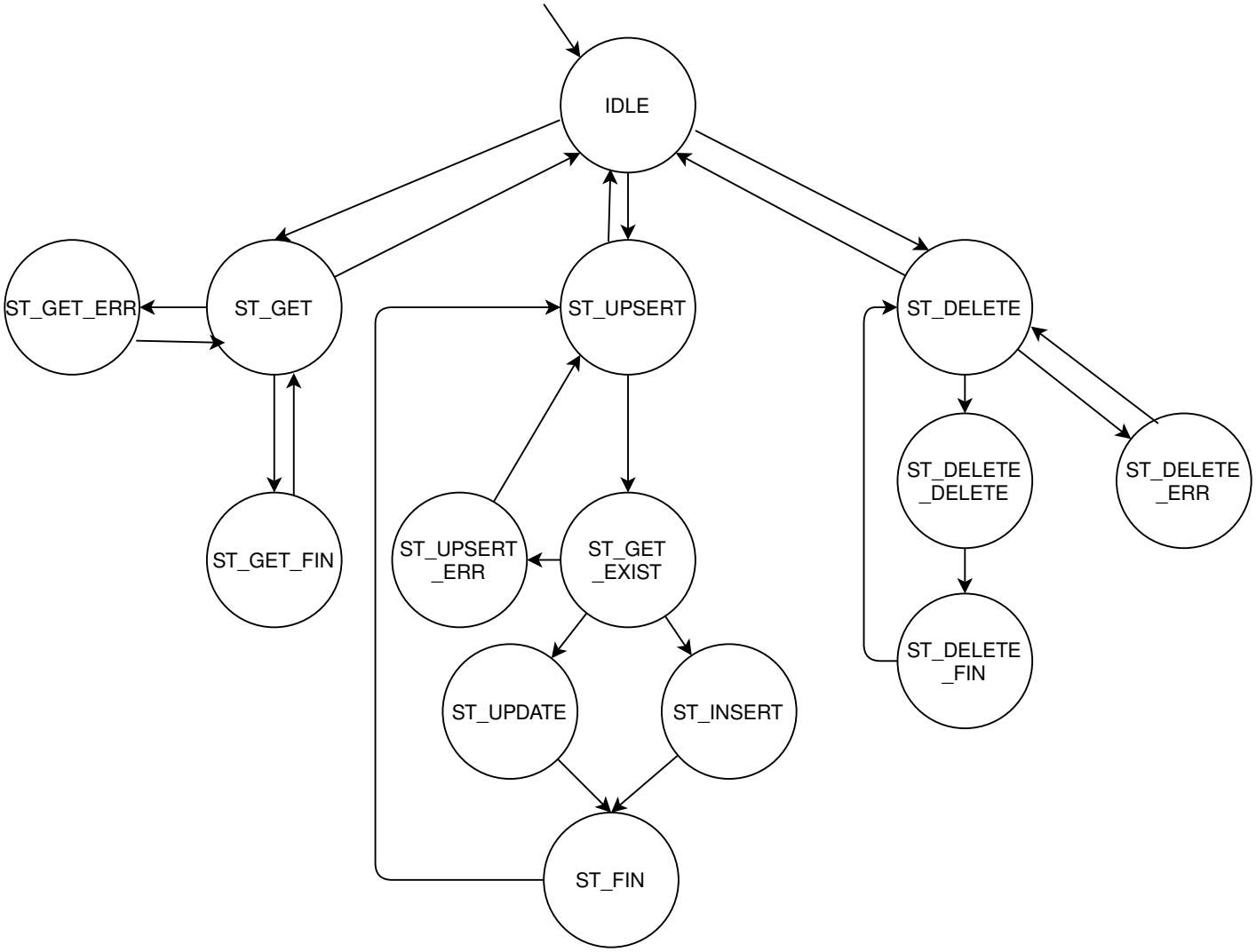

Abbildung 3: Initiale Statemachine

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

Während der Implementierungsphase 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.

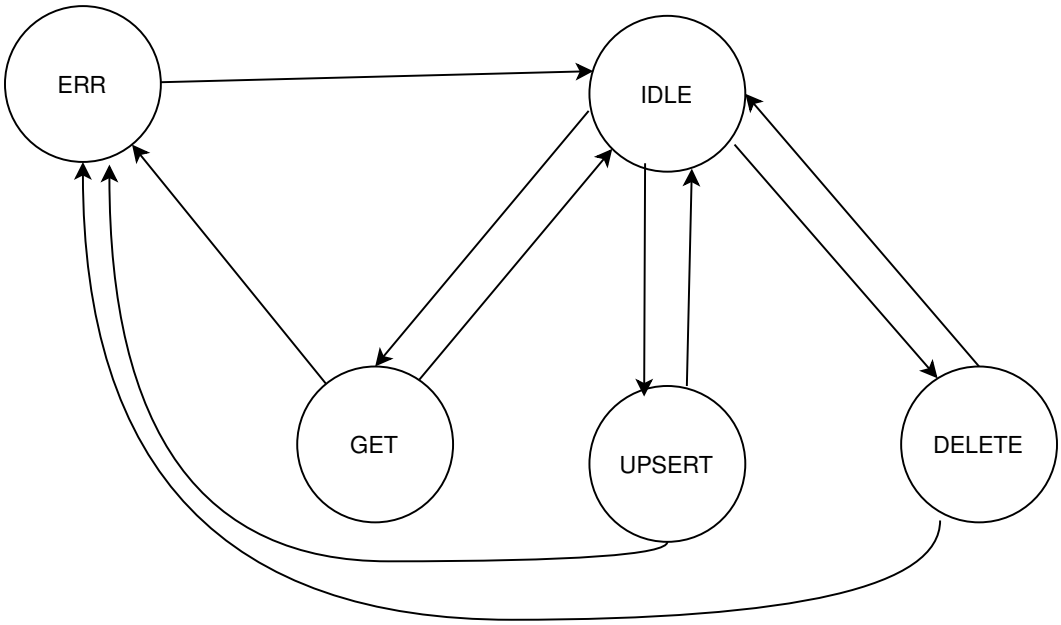

Abbildung 4: Statemachine

3.2 Taktzyklus Beispiele

Basierend auf der ursprünglichen Statemachine haben wir zur effizienteren 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.

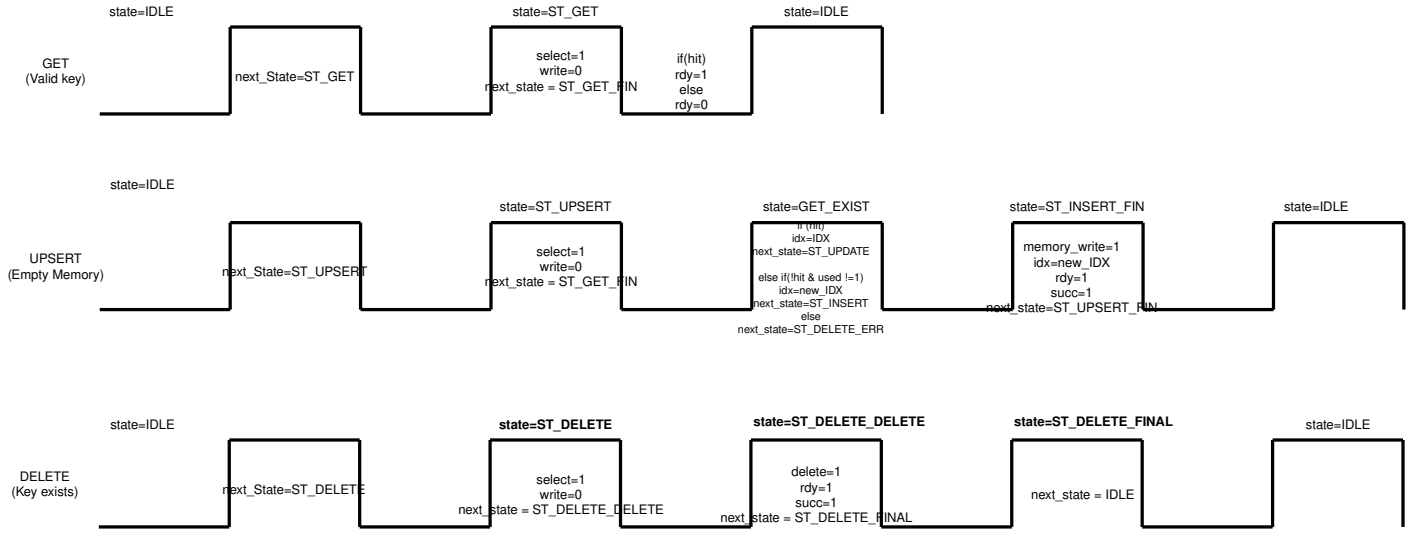

Abbildung 5: taktzyklus

Mithilde der Python-Tests (Cocotb) konnten die tatsächlichen Taktzyklen nach der Implementierung dargestellt werden. Siehe Kapitel Tests. Folgende Abbildungen zeigen Screenshots zu den einzelnen Operationen:

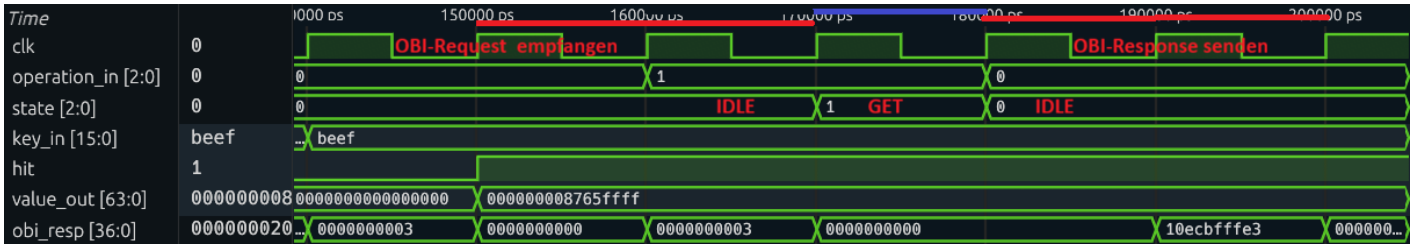

Abbildung 6: GET

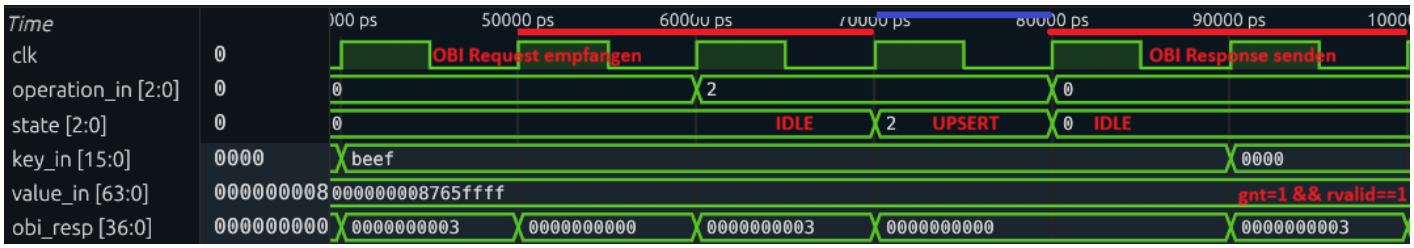

Abbildung 7: UPSERT (empty memory)

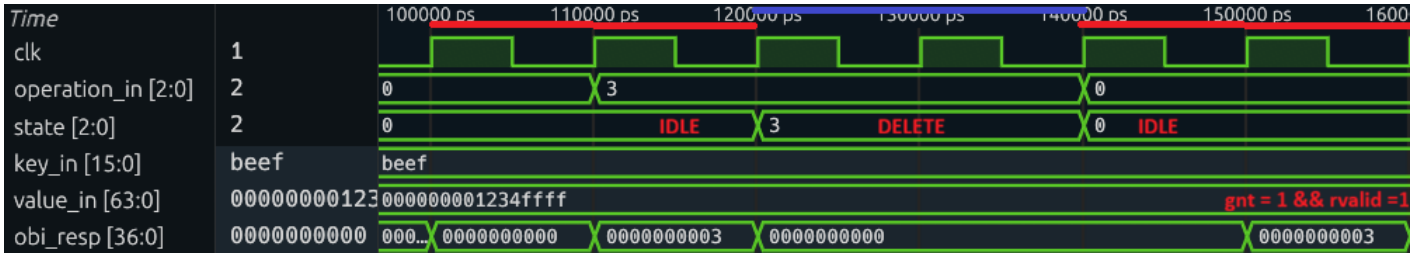

Abbildung 8: DELETE (key exists)

4 Implementationen

4.1 Memory

Ausarbeitung Philipp Hecht

Wie bereits zuvor beschrieben, haben wir uns für die Implementierung eines Key-Value-Stores entschieden, der im Wesentlichen auf einem einfachen Speicherblock basiert. Zunächst wurde für den Speicherblock definiert, dass er in der Lage sein soll,

1) Schlüssel-Wert Paare zu speichern
2) Schlüssel-Wert Paare zu löschen
3) Schlüssel-Wert Paare anhand von Schlüsseln zu lesen und auszugeben

Die Implementierung wurde hierfür in weitere kleinere Submodule hierarisch unterteilt. Zunächst wurde eine dynamisch ansteuerbarer Speicheeinheit implementiert, worauf im Nachfolgendem eingegangen wird. Darauf aufbauend wurde eine einzelne Speicherzelle definiert, welche die übergeordnete Funktionalität des Speicherns von Schlüssel-Wert Paaren bereitstellt. Hierauf setzend wurde ein kleiner Controller definiert, welcher N Schlüssel-Wert Paare in einem Array von Speicherzellen verwaltet und die zuvor beschreibenenen Funktionen bereitstellt.

4.1.1 Dynamic Register Array

Diese Komponente bildet die unterste Hierarchie-Ebene der Memory-Funktionalität und stellt einen einzelnen, konfigurierbaren Registerblock dar. Die Registerbreite wird durch den Parameter `LENGTH` zur Compile-Zeit definiert, wodurch eine flexible Anpassung an verschiedene Datenbreiten ermöglicht wird. Der Registerblock ist nach außen hin vollständig addressierbar durch standardisierte Steuersignale für Lese- und Schreibvorgänge.

Implementierung:

```
module dynamic_register_array #(
    parameter LENGTH
)(
  ....
);

    reg [LENGTH-1:0] registers;

    always_ff @(negedge clk or negedge rst_n) begin
        if (!rst_n) begin
            registers <=  '0;
        end else if (write_op) begin
            registers <= data_in;
        end
    end
    assign data_out = registers;
endmodule
```

Dieses Basis-Registermodul wird für jede Speicherzelle instanziiert und bildet die Grundlage für die höherrangigen Speicherfunktionen im Controller. Zur Optimierung des gesamten Caches beschlossen wir, dass die Registerblöcke nicht, wie üblich, auf der steigenenden Taktflanke beschrieben werden, sondern stattdessen auf der fallenden. Dadurch können wir die Daten bereits im nächsten Zyklus für Controller und die darüber ligendenden Blöcke bereitstellen, was es uns ermöglichte, die gesamte Speicherfunktionalität in nur einem Taktzyklus zu realisieren.

4.1.2 Memory Cell

Die Memory Cell ist die zweite Hierarchie-Ebene und kapselt eine einzelne Speicherzelle, welche Key-Value-Pair speichert. Dabei nutzt sie das zuvor beschriebenen Dynamic Register Array Modul für die Speicherung des Schlüssels als auch des Wertes:

```
module memory_cell #(
    parameter KEY_WIDTH,
    parameter VALUE_WIDTH
)(
    ...
    input logic [KEY_WIDTH-1:0] key_in,
    input logic [VALUE_WIDTH-1:0] value_in,

    output logic [KEY_WIDTH-1:0] key_out,
    output logic [VALUE_WIDTH-1:0] value_out,
    output logic used_out
);
    // Key Register
    dynamic_register_array #(.LENGTH(KEY_WIDTH)) key_reg (
        ...
    );

    // Value Register
    dynamic_register_array #(.LENGTH(VALUE_WIDTH)) value_reg (
        ...
    );

    // Used-Flag: Zelle ist gültig, wenn Key ungleich 0
    assign used_out = (key_out != '0);

endmodule
```

Durch das zuvorige Implementieren des Dynamic Register, können wir innerhalb der Zelle verschiedene Bit-Längen für die Schlüssel und Daten definieren.

Zusätzlich wurde ein "Used-Flag" implementiert, welches anzeigt, ob die Zelle aktuell gültige Daten enthält. Diese Funktionalität wird für den darüber ligenenden Controller benötigt. Da hierdurch bestimmt werden kann, ob der Cache vollständig gefüllt ist oder ob noch freie Zellen vorhanden sind. Zunächst bestand die Idee, die einzelnen Schlüssel zeitlich auslaufen zu lassen (ähnlich zu Redis). Aufgrund von Zeitmangel, wurde diese Funktionalität jedoch nicht implementiert, sodass als Verkürzung die Gültigkeit eines Schlüssel-Wert-Paares durch das Vorhandensein eines Schlüssels (key_out != 0) definiert wurde. Gleichzeitig beschlossen wir, dass die Übergeordnete Verwaltungslogik, welche Zellen aktuell *frei* sind innerhalb des Controllers stattfinden soll, was die Implementierung der Memory Einheit vereinfachte. Auch wollten wir nicht den Used-Wertes einer einzelnen Zelle über ein eigenes Register abbilden um damit auch hier eine Optimierung zu erreichen.

4.1.3 Memory Block

Der Memory Block ist die übergeordnete Kontrollschicht der einzelnen Speicherzellen. Gleichzeitig übernimmt dieser Block die Verwaltung von Befehlen des übergeordneten Controllers, welcher im Nachfolgenden Kapitel beschrieben wird.

Der Block instantiiert `NUM_ENTRIES` Speicherzellen und verwaltet diese als Array. Die Kontrolle über Schreib- und Löschvorgänge erfolgt durch gezieltes Aktivieren einzelner Zellen mittels Index-Signalen. Hierauf wird im Nachfolgendem näher eingegangen:

4.1.3.1 Einfügen von Schlüssel-Wert Paaren

Das Einfügen eines Schlüssel-Wert Paares wird durch ein Signal des Controllers ausgelöst. Zusammen mit dem Setzen des ausgewählten Indexes (Der Controller nutzt hierfür die *used* Signale der einzelnen Zellen (Hot-Wire)) wird die entsprechende Zelle aktiviert, um die Daten zu speichern. Bei der nächsten positiven Taktflanke werden die Daten in der Zelle gespeichert.

Als eine Optimierung wird die Löschoperation als Sonderfall eines Schreibvorgangs behandelt. Das bedeutet, dass beim Löschen eines Schlüssel-Wert Paares die Zelle mit einem Schlüssel von 0 beschrieben wird, wodurch sie als ungültig markiert wird.

```
    memory_cell #(
        .KEY_WIDTH(KEY_WIDTH),
        .VALUE_WIDTH(VALUE_WIDTH)
    ) temp (
        ...
        .write_op((write_in || delete_in) && index_in[i]),
        .key_in(delete_in ? '0 : key_in),
        .value_in(delete_in ? '0 : value_in),
        ...
    );
```

4.1.3.2 Abrufen von Werten anhand von Schlüsseln

Für das Abrufen von Schlüssel-Wert Einträgen empfängt der Memory Block einen Schlüssel als Input und durchsucht alle Speicherzellen nach einer Übereinstimmung. Dieser Prozess findet vollständig kombinatorisch im `always_comb` Block statt und liefert daher im selben Taktzyklus ein Ergebnis.

Die Such-Logik vergleicht den Input-Schlüssel mit allen gespeicherten Schlüsseln und gibt den zugehörigen Wert zurück. Dies wird im `always_comb` Block durch folgende Schritte realisiert:

```
// Schlüssel-basierter Zugriff: Alle Zellen nach Key durchsuchen
for (int i = 0; i < NUM_ENTRIES; i++) begin
    if (used_entries[i] && (cell_key_out[i] == key_in)) begin
        value_out_d = cell_value_out[i];    // Wert der gefundenen Zelle
        hit_d = '1;                          // Signalisiere erfolgreichen Match
        index_out_d = 1 << i;                // Gebe Position als One-Hot aus
    end
end
```

Bei einem erfolgreichen Match werden drei Ausgänge gesetzt: - `value_out_d`: Der gespeicherte Wert der gefundenen Zelle - `hit_d`: Ein Flag, das signalisiert, dass ein Match gefunden wurde - `index_out_d`: Die Position der gefundenen Zelle im One-Hot-Encoding Format

4.1.3.3 Löschen von Werten anhand von Schlüsseln

Wie bereits zuvor erwähnt, wird die Löschoperation als Sonderfall eines Schreibprozesses behandelt. Falls der Controller ein Löschsignal sendet, wird die gesamte Zellen auf *0* gesetzt:

```
memory_cell #(
    .KEY_WIDTH(KEY_WIDTH),
    .VALUE_WIDTH(VALUE_WIDTH)
) temp (
    ...
    .write_op((write_in || delete_in) && index_in[i]),
    .key_in(delete_in ? '0 : key_in),
    .value_in(delete_in ? '0 : value_in),
    ...
);
```

Nachfolgendes Architekturdiagramm zeigt den Aufbau des Memory Blockes und deren Teilkomponenten als auch die Signale, welche für die Interaktion mit dem übergeordneten Controller definiert wurden:

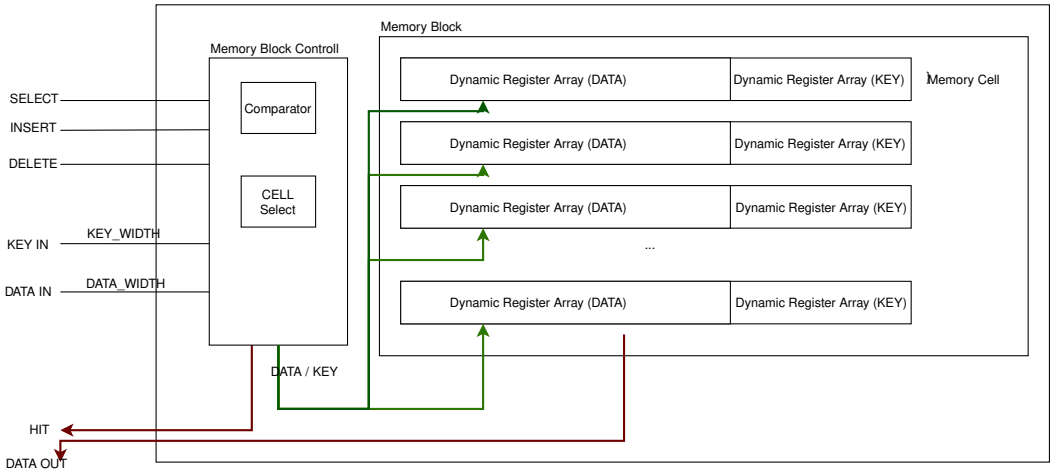

Abbildung 9: Memory Block Architektur

4.2 Controller

4.2.1 Main Controller

Ausarbeitung Luca Schmid

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 haben wir 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, haben wir für alle Sub-Controller ein einheitliches Interface-Konzept zur Kommunikation mit dem Main Controller entworfen:

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

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

Ausarbeitung Luca Pinnekamp

Für das Auslesen von Werten aus dem Cache haben wir die GET-Operation implementiert. Die zugehörige FSM (`get_fsm`) wird vom Main Controller aktiviert, sobald ein Lesezugriff angefordert wird.

Den Ablauf einer GET-Operation haben wir wie folgt gestaltet:

1. **Schlüsselsuche:** Um Latenzen zu minimieren, haben wir uns dazu entschieden, den zu suchenden Schlüssel nicht durch den Controller oder die `get_fsm` zu leiten. Stattdessen liegt dieser direkt am Interface an und wird von dort kontinuierlich an das Speichermodul übergeben. Das Speichermodul prüft daraufhin asynchron, ob ein entsprechender Eintrag im Cache existiert.
2. **Hit/Miss-Auswertung:** Das Speichermodul liefert ein `hit`-Signal an die `get_fsm` zurück. Da die Schlüssel und daraus resultierenden Werte direkt zwischen Interface und Memory Block übertragen werden muss die `get_fsm` lediglich die hit werte an den übergeordneten Controller bzw. das Interface weiterleiten. Zudem wird das `hit_valid` signal gesetzt um dem interface mitzuteilen, dass der Wert in das Kontrollregister übernommen werden kann.
3. **Abschluss:** Die `get_fsm` signalisiert dem Main Controller den Abschluss der Operation (`cmd.done = 1`), woraufhin dieser den Status und aktuellen Befehl auf die Standard Werte zurücksetzt und in den Idle Zustand wechselt.

4.2.3 UPSERT

Ausarbeitung Luca Schmid

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

Ausarbeitung Philipp Hecht

Die DELETE FSM verwaltet den Löschprozess durch drei Zustände:

1. **DEL_ST_START**: In diesem Zustand wird keine Operation ausgeführt; das Modul wartet auf die Aktivierung durch den Controller. Sobald das Modul aktiviert wird (dediziertes Steuersignal), springt die State Machine in den nächsten Zustand, um die Löschoperation einzuleiten.
2. **DEL_ST_DELETE**: Die Statemachine behandelt das Löschen eines Schlüssels aus dem Memory Block. Aus dem eingehenden HIT-Signal wird vom Memory Block erkannt, dass der Schlüssel abgespeicher worden ist. Sollte dies der Fall sein, wird ein Löschsignal an den Memory Block gesendet, welcher, in Kombination mit dem bereits anliegendem Schlüssel, spätestens zur negativen Taktflanke vom Memory Block durchgeführt wird.
3. **DEL_ST_ERROR**: Wenn der Memory Block keine Übereinstimmung zum zu löschenden Schlüssel findet (Hit-Signal bleibt Low), wechselt die FSM in diesen Fehlerzustand. Der Substate übermittelt an den übergeordneten Controller, dass die Löschoperation nicht erfolgreich war.

Beide Zustände DEL_ST_DELETE und DEL_ST_ERROR wechseln zur nächsten positiven Flanke zurück in den DEL_ST_START Zustand, um die nächste Löschoperation entgegenzunehmen.

Dieses Timing ermöglicht es, dass die DELETE-Operation in nur zwei Taktzyklen abgeschlossen wird. Die zuvor beschriebene `always_comb`-Logik des Memory Blockes ermöglicht es direkt einen Löschvorgang abzuschließen.

4.3 Obi interface

Als übergeordneten Block wurde für die Anbindung des Caches eine OBI (Open Bus Interface) Schnittstelle implementiert. Diese ermöglicht es, den Cache über ein standardisiertes Protokoll zu steuern. Weiteres wird im Nachfolgendem Kapitel OBI beschrieben.

5 CROC

Ausarbeitung Luca Pinnekamp

Das CROC (Custom RISC-V Open-source Core) System-on-Chip (SoC) dient als Zielplattform für die Integration unseres Hardware-Redis-Caches. Wir haben uns für diese Plattform entschieden, da sie die notwendige Infrastruktur bietet, um unsere Erweiterung in einem realistischen System zu evaluieren.

5.1 CROC Architektur

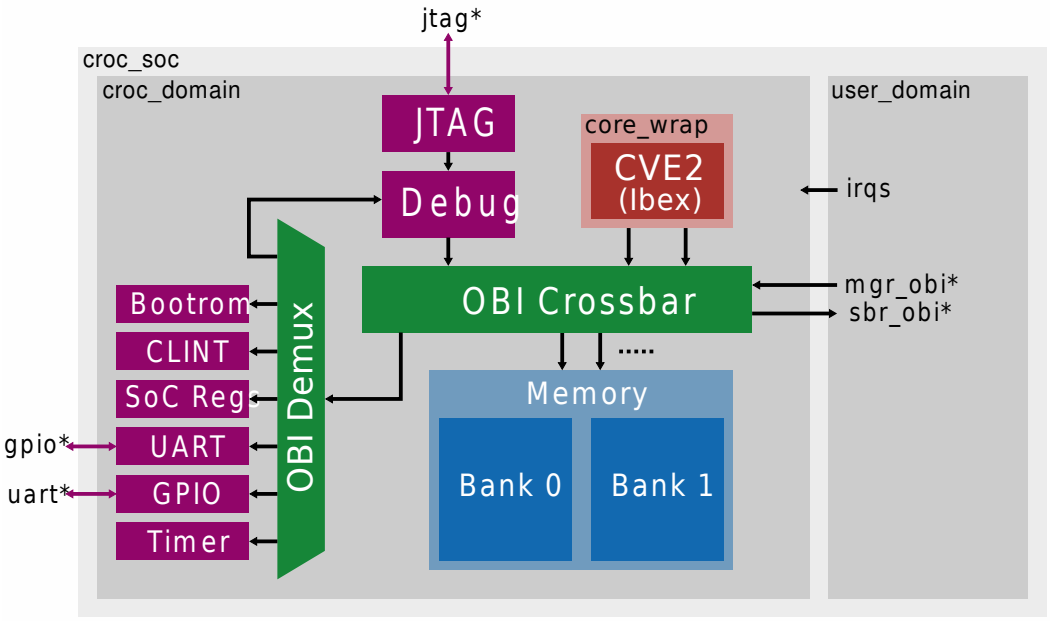

Abbildung 10: CROC SoC Architektur

Die Architektur des CROC SoCs ist hierarchisch in verschiedene Domänen unterteilt, um eine klare Trennung zwischen dem Kernsystem und benutzerdefinierten Erweiterungen zu gewährleisten:

- **croc_soc**: Die oberste Ebene des Systems, die alle Subsysteme und externen Schnittstellen zusammenfasst.
- **croc_domain**: Beinhaltet den eigentlichen RISC-V Prozessorkern, den primären Speicher sowie grundlegende Peripheriegeräte (UART, Timer, etc.).
- **user_domain**: Ein dedizierter Bereich für benutzerdefinierte Hardware. Dieser Bereich ist über einen OBI Bus an die `croc_domain` angebunden. Hier haben wir unseren Redis Cache integriert.

5.2 Implementation bei uns im Projekt

Um den Redis Cache in das CROC SoC zu integrieren, mussten wir spezifische Anpassungen in der `user_domain` und den zugehörigen Konfigurations-Packages vornehmen. Diese waren nötig um unseren OBI Subordinate an den bestehenden OBI Crossbar anzuschließen.

5.2.1 Anpassungen im `user_pkg`

Das `user_pkg` definiert die Speicherarchitektur und die Adressräume der benutzerdefinierten Peripherie. Folgende Änderungen haben wir hier vorgenommen:

1. **Adressraum-Definition**: Wir haben dem Redis Cache einen festen Adressbereich im Memory Map des SoCs zugewiesen. Dazu definierten wir eine Basisadresse (`UserRedisCacheAddrOffset`) und die Größe des Adressraums (`UserRedisCacheAddrRange`).
2. **Erweiterung der Peripherie-Anzahl**: Wir haben die Konstante für die Anzahl der Subordinates im User-Interconnect (`NumUserDomainSubordinates`) erhöht, um den neuen IP-Core aufnehmen zu können.
3. **Adress-Dekodierung**: Die Adress-Dekodierungsregeln (`user_addr_map`) für den Crossbar haben wir um den Bereich des Redis Caches erweitert, damit Speicherzugriffe des Prozessors korrekt an unseren IP-Core geroutet werden.

```
localparam bit [31:0] UserRedisCacheAddrOffset = UserRomAddrOffset + UserRomAddrRange;
localparam bit [31:0] UserRedisCacheAddrRange  = 32'h0000_1000;

// Address rules given to address decoder
localparam croc_pkg::addr_map_rule_t [NumDemuxSbrRules-1:0] user_addr_map = '{
  // 1: ROM
  '{ idx: UserRom,
     start_addr: UserRomAddrOffset,
```

```
      end_addr: UserRomAddrOffset + UserRomAddrRange },
    // 2: RedisCache
    '{ idx: UserRedisCache,
      start_addr: UserRedisCacheAddrOffset,
      end_addr: UserRedisCacheAddrOffset + UserRedisCacheAddrRange }
};
```

5.2.2 Anpassungen in der `user_domain`

In der `user_domain` erfolgt die eigentliche Instanziierung und Verdrahtung der Hardware-Module. Hierbei wird der Redis Cache als Subordinate in den OBI-Bus eingehängt:

1. **Signal-Deklaration**: Zunächst haben wir die OBI-Request- und Response-Signale (`user_redis_cache_obi_req`, `user_redis_cache_obi_rsp`) für den Cache deklariert und mit dem Demultiplexer (`all_user_sbr_obi_req`, `all_user_sbr_obi_rsp`) verbunden.
2. **Instanziierung des Redis Cache**: Anschließend haben wir das Top-Level-Modul unseres Redis Caches in der `user_domain` instanziiert.
3. **OBI-Parameter**: Um eine korrekte implementation des OBI Interface zu ermöglichen werden hier die vom croc SOC definierten Parameter für die `ObiCfg`, `obi_req_t` und `obi_rsp_t` übergeben um als Datentypen bzw. für die Konfiguration der korrekten Busbreite im Redis Cache verwendet zu werden.
4. **OBI-Schnittstellen-Verbindung**: Die zuvor deklarierten OBI-Signale haben wir an die entsprechenden Ports des Caches angeschlossen.

```
// OBI bus to RedisCache
sbr_obi_req_t user_redis_cache_obi_req;
sbr_obi_rsp_t user_redis_cache_obi_rsp;

// Fanout into more readable signals
assign user_redis_cache_obi_req             = all_user_sbr_obi_req[UserRedisCache];
assign all_user_sbr_obi_rsp[UserRedisCache] = user_redis_cache_obi_rsp;

// ... (Demultiplexer Logik) ...

// RedisCache Subordinate
redis_cache #(
  .ObiCfg      ( SbrObiCfg     ),
  .obi_req_t   ( sbr_obi_req_t ),
  .obi_rsp_t   ( sbr_obi_rsp_t )
) i_user_redis_cache (
  .clk (clk_i),
  .rst_n (rst_ni),
  .obi_req_i  ( user_redis_cache_obi_req ),
  .obi_resp_o  ( user_redis_cache_obi_rsp )
);
```

Durch diese strukturierte Integration ist der Redis Cache nun als Memory-Mapped I/O (MMIO) Gerät für den RISC-V Prozessor sichtbar und kann über die von uns entwickelte C-Bibliothek angesteuert werden.

6 OBI

Ausarbeitung Philipp Hecht

6.1 Einleitung und Motivation

Ausarbeitung Philipp Hecht

Die ursprüngliche Idee war es, den Cache über *Custom-Commands* innerhalb einer RISC-V CPU anzusteuern. Hierrüber wäre es somit möglich gewesen, die Cache Operationen direkt als Assembler Instruktionen zu implementieren. Auf Empfehlung des Lehrpersonals, wurde die Entscheidung getroffen, die Anbindung des Caches über Memory Mapping zu realisieren. Als mögliche Protokolle hierfür fanden wir das AXI als auch das OBI (Open Bus Interface) Protokoll. Nach ersten Versuchen

zur Implementierung des AXI Protokolls, entschieden wir uns auf Anraten des Lehrpersonals für die Implementierung des OBI Protokolles, da dieses bereits als vorgefertiges Projekt bereits (vgl. Kapitel Croc) vorliegt.

Architekturell entschlossen wir uns, den OBI Slave nicht direkt in den Cache Controller zu implementieren sondern stattdessen, diesen als separates Modul aufzusetzten. Dies ermöglichte uns einerseits die aktuelle Implementierung des Caches nicht zu verändern. Der Cache erhält weiterhin die gleichen Signale wie zuvor, was die Komplexität der Implementierung reduzierte. Andererseits ermöglichte eine separate Implementierung dediziertes Testen und Debugging des OBI Slaves.

6.2 OBI Protokoll

Ausarbeitung Luca Schmid

6.2.1 Register

Mit dem OBI Protokoll ermöglichen wir der CPU über Memory Mapping direkt mit dem Controller zu kommunizieren. Hierfür definierten wir 3 Registertypen, die über das OBI Protokoll angesprochen werden können:

	Daten	Schlüssel	Op-Code
Größe	64 (8 Bytes)	32 (4 Bytes)	32 (4 Bytes)
Offset	0	8	12

Die Größe der Register ist in der cache_cfg_pkg konfigurierbar. Die Offset Werte leiten sich aus den den Größen der Key und Value Register Modulo 32 ab. In den OBI Nachrichten wird die Speicheraddresse(Basis + Offset) mitgegeben. Über dieses Offset werden die Daten in die richtigen Register geschrieben bzw. können darüber ausgelesen werden.

6.2.2 OBI Request

Über einen OBI Request sendet die Applikation (Master) Daten an den Cache. Hierfür sieht das OBI Protokoll eine bestimmte Struktur der Nachricht vor. Nachfolgende Tablle beschreibt die für den Cache relevanten Felder des OBI Requests:

Feld in obi_req_t	Bit-Breite	Beschreibung
addr	32	Adresse von der CPU an der die Daten geschrieben werden sollen
we	1	Write Enable: 1 führt einen Schreiben aus, 0 führt ein Lesen aus
wdata	32	Write Data: Die Daten, die am Offset der Addresse eingeschrieben werden sollen
aid	1	Address ID: Eine ID für die Transaktion
req	1	Request: Das Handshake-Signal, welches der Master zum Start einer Transaktion sendet

*für die Dokumenatiion nicht relevanten felder wurden hier ausgelassen

Eine OBI Transkation besteht dabei immer aus einem OBI Request, sowie der zugehörigen Response.

6.2.3 OBI Response

Die OBI Response wird von der Hardware an den Master versendet. Hierrüber können die Werte des angefragten Register an die Applikation zurückgegeben werden. Nachfolgende Tablle beschreibt die für den Cache relevanten Felder des OBI Responses:

Feld in obi_rsp_t	Bit-Breite	Beschreibung
rdata	32	Read Data: Daten, die vom Cache zurück an die Applikation versendet werden

Feld in obi_rsp_t	Bit-Breite	Beschreibung
rid	1	Response ID: Spiegelt die aid aus dem Request wider, um Antworten zuzuordnen
err	1	Error: Einfaches Flag Signal, welches der Applikation andeutet, dass die Ausführung nicht erfolgreich war
gnt	1	Grant: Handshake-Signal mit dem der Slave bestätigt, dass er die Anfrage verarbeitet hat
rvalid	1	Response Valid: Wird 1, wenn die zurückgegebenen Daten in r.rdata gültig sind.

*für die Dokumenatiion nicht relevanten felder wurden hier ausgelassen

6.3 Implementierung des OBI Slaves

Ausarbeitung Philipp Hecht

Zunächst versuchten wir, den OBI Slave eigenständig zu implementieren. Vorwegnehmend war dieser Weg zur Implementierung nicht erfolgreich. Dennoch soll hier der Weg zur erfolgreichen Implementierung beschrieben werden, um damit das generelle Konzept hinter unserer Implementierung zu verdeutlichen:

Die Risc-V CPU kommuniziert über Memory Mapping mit dem Cache. Hierfür werden die OBI Requests und Responses über die entsprechenden Signale gesendet. Der vom Croc bereitgestellte RISC-V Core arbeitet dabei auf einer 32-Bit Architektur, weshalb die Datenbreite eines einezelnen OBI Requests bzw. Response auf 32 Bit limitiert ist. Gleichzeitig arbeitet der Cache mit einer Datenlänge, welche höher als die 32 Bit ist. Dies stellte die erste Herausforderung für die Interface Implementierung dar.

Aus Sicht des OBI Protokolles, wird dieses Problem darüber gelöst, dass mehrere, hintereianderliegende Adressen für die gesamte Datenübertragung genutzt werden. Es ist dann Aufgabe der Applikation die Daten entsprechend über mehrere Aufrufe an die zugehörigen Register zu schreiben (siehe hierzu das Kapitel C Lib).

Pseudo Code hierfür würde in etwa wie folgt aussehen:

```
void upsert_to_cache(uint32_t base_address, uint64_t data, uint32_t key) {
    // Aufteilen der 64-Bit Daten in zwei 32-Bit Teile
    uint32_t lower_data = (uint32_t)(data & 0xFFFFFFFF);
    uint32_t upper_data = (uint32_t)((data >> 32) & 0xFFFFFFFF);

    write_to_obi(base_address, lower_data);
    write_to_obi(base_address + 4, upper_data);
    write_to_obi(base_address + 8, key);
    write_to_obi(base_address + 12, OP_CODE_UPSERT);
}
```

Darauf aufbauend muss das OBI Interface Modul über einen State Machine arbeiten, um eingehende OBI Transaktionen zu verarbeiten und die entsprechenden Daten in ein temporäres Register zu schreiben. Als *Go* für das Übermitteln der Daten an den Cache Controller definierten wir, dass sobald der Master den Operationscode über das entsprechende Register schreibt, die Daten an den Cache Controller weitergeleitet werden. Die State Machine hatte dabei die folgenden Zustände:

- IDLE: In diesem Zustand wartet die State Machine auf einen OBI Request. Sobald ein Operationscode über das entsprechende Register geschrieben wird, übermittelt das Interface die Daten an den Controller und wechselt in einen WAIT_FOR_CONTROLLER Zustand.
- WAIT_FOR_CONTROLLER: In diesem Zustand wartet die State Machine auf eine Rückmeldung des Cache Controllers. Sobald der Controller die Daten verarbeitet hat, wechselt die State Machine zurück in den IDLE Zustand und ist bereit für die nächste Transaktion. Während dieser Phase blockiert die State Machine weitere OBI Requests und setzt `rvalid` auf 0, um der Applikation mitzuteilen, dass die Daten noch nicht zurückgegeben werden können.
- COMPLETE: In diesem Zustand werden die Daten an die Applikation zurückgegeben. Sobald die Daten zurückgegeben wurden, wechselt die State Machine zurück in den IDLE Zustand.

Es zeigte sich jedoch, dass diese Art der Implementierung nicht sonderbar kombinierbar mit dem Ablauf des OBI Protokolles einhergehend war. Erste Implementierungen blockierten dabei die gesamte OBI Crossbar und blockierten damit nicht nur den Cache als auch die CPU selbst (vermutlich).

Nach Absprache mit dem Lehrpersonal überarbeiteten wir die Implementierung dahingehend um, dass die Statemaschine nicht mehr über einen internen Zustandsautomaten verfügt, sondern *OBI nativ* jegliche Anfragen immer annimmt und direkt darauf reagiert. Das Interface *blockiert* nicht mehr während der Verarbeitung einer Anfrage, sondern nimmt direkt die nächste Anfrage an. Hierdurch reduziert sich die Komplexität des OBI Interfaces. Gleichzeitig ist es damit Aufgabe des Masters die Anfragen in sequenzieller Reihenfolge zu stellen. Das Lesen von Daten während der Controller noch mit der Verarbeitung einer vorherigen Anfrage beschäftigt ist, führt zu fehlerhaften Daten.

Zunächst versuchten wir, den OBI Slave eigenständig zu implementieren. Vorwegnehmend war diese Weg zur Implementierung nicht erfolgreich. Dennoch soll hier der Weg zur erfolgreichen Implementierung beschrieben werden, um damit das generelle Konzept hinter unserer Implementierung zu verdeutlichen:

Die Risc-V CPU kommuniziert über Memory Mapping mit dem Cache. Hierfür werden die OBI Requests und Responses über die entsprechenden Signale gesendet. Der vom Croc bereitgestellte RISC-V Core arbeitet dabei auf einer 32-Bit Architektur, weshalb die Datenbreite eines einezelnen OBI Requests bzw. Response auf 32 Bit limitiert ist (gilt für wdata, addr, rdata). Gleichzeitig arbeitet der Cache mit einer Datenlänge, welche höher als die 32 Bit ist. Dies stellte die erste Herausforderung für die Interface Implementierung dar.

Aus Sicht des OBI Protokolles, wird dieses Problem darüber gelöst, dass mehrere, hintereianderliegende Adressen für die gesamte Datenübertragung genutzt werden. Es ist dann Aufgabe der Applikation die Daten entsprechend über mehrere Aufrufe an die zugehörigen Register zu schreiben (siehe hierzu das Kapitel zur C Bibliothek).

Pseudo Code hierfür würde in etwa wie folgt aussehen:

```
void upsert_to_cache(uint32_t base_address, uint64_t data, uint32_t key) {
    // Aufteilen der 64-Bit Daten in zwei 32-Bit Teile
    uint32_t lower_data = (uint32_t)(data & 0xFFFFFFFF);
    uint32_t upper_data = (uint32_t)((data >> 32) & 0xFFFFFFFF);

    write_to_obi(base_address, lower_data);
    write_to_obi(base_address + 4, upper_data);
    write_to_obi(base_address + 8, key);
    write_to_obi(base_address + 12, OP_CODE_UPSERT);
}
```

Darauf aufbauend muss das OBI Interface Modul über einen State Machine arbeiten, um eingehende OBI Transaktionen zu verarbeiten und die entsprechenden Daten in ein temporäres Register zu schreiben. Als *Go* für das Übermitteln der Daten an den Cache Controller definierten wir, dass sobald der Master den Operationscode über das entsprechende Register schreibt, die Daten an den Cache Controller weitergeleitet werden. Die State Machine hatte dabei die folgenden Zustände:

- IDLE: In diesem Zustand wartet die State Machine auf einen OBI Request. Sobald ein Operationscode über das entsprechende Register geschrieben wird, übermittelt das Interface die Daten an den Controller und wechselt in einen WAIT_FOR_CONTROLLER Zustand.
- WAIT_FOR_CONTROLLER: In diesem Zustand wartet die State Machine auf eine Rückmeldung des Cache Controllers. Sobald der Controller die Daten verarbeitet hat, wechselt die State Machine zurück in den IDLE Zustand und ist bereit für die nächste Transaktion. Während dieser Phase blockiert die State Machine weitere OBI Requests und setzt `rvalid` auf 0, um der Applikation mitzuteilen, dass die Daten noch nicht zurückgegeben werden können.
- COMPLETE: In diesem Zustand werden die Daten an die Applikation zurückgegeben. Sobald die Daten zurückgegeben wurden, wechselt die State Machine zurück in den IDLE Zustand.

Es zeigte sich jedoch, dass diese Art der Implementierung nicht sonderbar kombinierbar mit dem Ablauf des OBI Protokolles einhergehend war. Erste Implementierungen blockierten dabei die gesamte OBI Crossbar und blockierten damit nicht nur den Cache als auch die CPU selbst (vermutlich).

Nach Absprache mit dem Lehrpersonal überarbeiteten wir die Implementierung dahingehend um, dass die Statemaschine nicht mehr über einen internen Zustandsautomaten verfügt, sondern *OBI nativ* jegliche Anfragen immer annimmt und direkt darauf reagiert. Das Interface *blockiert* nicht mehr während der Verarbeitung einer Anfrage, sondern nimmt direkt die nächste Anfrage an. Hierdurch reduziert sich die Komplexität des OBI Interfaces. Gleichzeitig ist es damit Aufgabe des Masters die Anfragen in sequenzieller Reihenfolge zu stellen. Das Lesen von Daten während der Controller noch mit der Verarbeitung einer vorherigen Anfrage beschäftigt ist, führt zu fehlerhaften Daten.

6.3.1 Aktuelle Implementierung

Ausarbeitung Luca Pinnekamp

Die finale Architektur des OBI-Interfaces ist stark an die Implementierung des UART OBI Interfaces aus dem Croc SoC angelehnt. Sie realisiert eine direkte Register-Schnittstelle, die als Bindeglied fungiert und von zwei Seiten unabhängig aktualisiert werden kann:

- **Updates durch die CPU (OBI-Seite):** Die CPU schreibt über reguläre OBI-Requests in die Register. Das Interface decodiert die Zieladresse und aktualisiert das entsprechende 32-Bit-Wort im `DAT`-, `KEY`- oder `CTR`-Register. Dabei werden die Byte-Enable-Signale (`be`) des OBI-Protokolls respektiert, sodass auch einzelne Bytes innerhalb eines Wortes gezielt beschrieben werden können (Auch wenn dies hier nicht unbedingt nötig ist). Eine Änderung der Operation im `CTR`-Register löst dabei die eigentliche Operation im Cache-Controller aus.
- **Updates durch den Cache-Controller (Interne Seite):** Die aktuellen Werte der Register werden kontinuierlich über interne Signale `reg_read_o` (`dat`, `key`, `operation`) an den Controller ausgegeben. Der Cache-Controller meldet Ergebnisse über dedizierte interne Signale `reg_write_i` (`dat`, `busy`, `hit`, `operation`) an das Interface zurück. Jedes Feld verfügt über ein eigenes Valid-Signal (`data_valid`, `busy_valid`, `hit_valid`, `operation_valid`), welches bestimmt, ob der entsprechende Wert im Register aktualisiert werden soll. Bei einem erfolgreichen `GET`-Befehl setzt der Controller beispielsweise `data_valid` auf 1 und überschreibt das `DAT`-Register mit dem gefundenen Wert aus dem Cache. Gleichzeitig werden die Status-Bits über `hit_valid` und `busy_valid` im `CTR`-Register aktualisiert.

Durch diese beidseitige Aktualisierung dient das Interface als reiner Datenspeicher. Die Synchronisation erfolgt ausschließlich über die Software, welche das `busy`-Bit auslesen muss, um festzustellen, wann der Controller seine Updates abgeschlossen hat.

7 FPGA

Ausarbeitung Luca Pinnekamp

Die Implementierung und Synthese unseres Redis Caches auf einem FPGA (Field Programmable Gate Array) ist ein zentraler Bestandteil dieses Projekts, um die Hardware-Beschleunigung in der Praxis zu evaluieren. Als Zielplattform haben wir uns für das **Digilent Genesys 2 Board** entschieden, da das CROC SoC bereits erfolgreich auf diesem Board getestet wurde und wir somit auf eine funktionierende Basis aufbauen konnten.

7.1 Synthese und Mapping

Für die Synthese des SystemVerilog-Codes verwenden wir Xilinx Vivado. Der Code wird zunächst analysiert und in eine Netzliste übersetzt, die aus logischen Gattern und Flip-Flops besteht. Anschließend erfolgt das Mapping auf die spezifischen Ressourcen des Kintex-7 FPGAs (z.B. LUTs, Flip-Flops und DSP-Slices).

Im Gegensatz zu herkömmlichen Cache-Implementierungen, die häufig auf dedizierte Block RAMs (BRAMs) zurückgreifen, haben wir unseren Redis Cache so entworfen, dass die Speicherung der Cache-Einträge (Keys und Values) primär in verteilten Logikressourcen (Distributed RAM / LUT-RAM) oder Registern erfolgt. Dies ermöglicht uns eine hochgradig parallele Architektur und schnelle Zugriffszeiten, erfordert jedoch eine sorgfältige Konfiguration der Speichergröße (`NUM_ENTRIES`), um die verfügbaren Logikressourcen (LUTs) des Genesys 2 Boards nicht zu überschreiten.

7.2 Integration und Test auf dem Genesys 2

Nach der erfolgreichen Synthese und dem Routing generieren wir den Bitstream. Dieser wird anschließend über einen USB-Stick direkt auf das Genesys 2 Board geladen, um das FPGA zu konfigurieren.

Sobald das FPGA mit dem CROC SoC und unserem integrierten Redis Cache konfiguriert ist, laden wir den C-Code für die Testprogramme über OpenOCD und GDB auf den RISC-V Prozessor. Ein großer Vorteil des Genesys 2 Boards ist hierbei, dass das SoC direkt den integrierten JTAG-USB-Port des Boards nutzt. Dadurch benötigen wir keinen zusätzlichen Raspberry Pi oder externe JTAG-Adapter für das Flashen und Debuggen. Über die UART-Schnittstelle des Genesys 2 Boards können wir dann die Ausführung der Programme überwachen und die Ergebnisse auf einem Host-PC auswerten.

8 C Lib

Ausarbeitung Luca Pinnekamp

Um die Interaktion mit dem Hardware-Redis-Cache aus einer Software-Umgebung heraus zu vereinfachen, haben wir eine dedizierte C-Bibliothek entwickelt. Diese Bibliothek abstrahiert die komplexen Hardware-Zugriffe über das OBI-Interface und stellt dem Entwickler eine einfache und intuitive API zur Verfügung.

8.1 Architektur der Bibliothek

Die C-Bibliothek basiert auf dem Prinzip des Memory-Mapped I/O (MMIO). Der Hardware-Cache ist in den Adressraum des Prozessors eingeblendet. In der Bibliothek haben wir Zeiger auf die spezifischen Basisadressen der Cache-Register (für Key, Value Low/High und Operation/Control) definiert. Die Basisadresse `REDIS_CACHE_BASE_ADDR` wird dabei zentral in der `config.h` konfiguriert.

8.2 Kernfunktionen

Die API bietet Funktionen für die grundlegenden Cache-Operationen und gibt jeweils einen Statuscode (`REDIS_CACHE_STATUS_OK`, `REDIS_CACHE_STATUS_MISS` oder `REDIS_CACHE_STATUS_ERR`) zurück:

- `redis_cache_get(key, &value_out)`: Schreibt den Schlüssel in das Key-Register, löst die GET-Operation aus und wartet auf den Abschluss. Bei einem Hit wird der 64-Bit Wert aus den Value-Registern gelesen und in `value_out` gespeichert.
- `redis_cache_upsert(key, value)`: Schreibt den 64-Bit Wert in die Value-Register, den Schlüssel in das Key-Register und triggert die UPSERT-Operation im Operations-Register.
- `redis_cache_delete(key)`: Übergibt den Schlüssel und startet die DELETE-Operation.

8.2.1 Beispiel: Werte schreiben und lesen

```c
#include "redis_cache.h"

uint32_t key = 0x11111111u;
uint64_t value = 0x1122334455667788ULL;

// Wert in den Cache schreiben
int status = redis_cache_upsert(key, value);
if (status != REDIS_CACHE_STATUS_OK) {
    // Fehlerbehandlung: Schreiben fehlgeschlagen
    return -1;
}

// Wert aus dem Cache lesen
uint64_t read_value = 0;
status = redis_cache_get(key, &read_value);
if (status == REDIS_CACHE_STATUS_ERR) {
    // Fehlerbehandlung: Hardware-Fehler oder Timeout
    return -1;
} else if (status == REDIS_CACHE_STATUS_MISS) {
    // Schlüssel nicht im Cache gefunden, z.B. Standardwert setzen
    read_value = 0;
}

// read_value enthält nun 0x1122334455667788ULL (oder 0 bei Miss)
```

8.3 Einbindung in Croc

Wir haben die C-Bibliothek so konzipiert, dass sie nahtlos in das Software-Ökosystem des CROC SoCs integriert werden kann. Sie wird als statische Bibliothek kompiliert und gegen die Anwendungssoftware gelinkt. Durch die Verwendung von standardisierten Datentypen und einer klaren Trennung von Hardware-spezifischen Adressen (die über Header-Dateien konfiguriert werden) ist die Bibliothek portabel und leicht anpassbar.

9 Backend

Ausarbeitung Philipp Hecht

Wie bereits erwähnt wurden wie in dem gezeigtem Beispielprojektes mit Submodulen gearbeitet, was es uns ermöglichte die einzelnen Komponenten (Memory Block, Cache Controller, OBI Interface) unabhängig voneinander zu entwickeln und zu testen. In diesem Kapitel soll es darüber hinausgehend auf das Bauen der einzelnen Module sowie das Ausführen der gesamten Backend Pipeline eingegangen werden.

Ein weiterer Vorteil der gesamten Modularisierung der Komponenten war die Möglichkeit, die einzelnen Module unabhängig voneinander zu bauen. Hierfür wurden in ähnlicher Weise wie vom Lehrpersonal bereitgestellt, *Makefile Targets* für das ausführen der Librelane Pipeline erstellt. Jedes Target baut ausgehend von einer `config.yaml` Datei die entsprechende Komponente. Auf unterster Ebene werden die jeweiligen Targets rekursiv aufgerufen:

Darüber hinaus wurde für das Tapeout des gesamten Systemes (inklusive der Croc Umgebung) auf Empfehlung des Lehrpersonals ein weiteres Code Repository erstellt. Diese beinhaltet die gesamte Croc Umgebung, die Anbindung an unser System sowie unsere eigene Implementierung (vgl. Kapitel CROC).

Das Croc Repository arbeitet zum Tapeout ihrer Implementierung mit Bender. Zur Transition zur unserer eigenen Tapeout Pipeline wurde zunächst versucht, über die Bender Pipeline eine vollumfängliche Liste aller notwendigen Verilog Dateien zu generieren. Die Idee war, diese in einer Librelane Konfigurationsdatei zu überführen um, darauf aufbauend, den gesamten SoC über die vom Lehrpersonal vorgestellten Tools zu bauen.

Nach mehreren gescheiterten Versuchen und Rücksprache mit dem Lehrpersonal wurden wir auf einen bereits existierenden Fork des Projektes verwiesen, welcher die Überführung uns abnimmt. Mit diesem Fork begannen wir das Tapeout Repository neu aufzusetzen und unsere Änderungen einzupflegen.

Aus zeitlichen Gründen durch Nahekommen der Abgabe konnten wir allerdings die komplette Transistion und Anpassung unserer Makefile Targets nicht mehr durchführen und testen. Stattdessen wurde nur ein Bau Prozess gestartet, welches nicht mit Makros arbeitet, sondern alle Verilog Dateien zusammen kompiliert.

Nach dem erfolgreichen Bau des gesamten Systems, konnten wir uns zuletzt das Endergebnis unserer Implementierung anschauen. Nachfolgende Abbildung zeigt die generierte GDSII Datei:

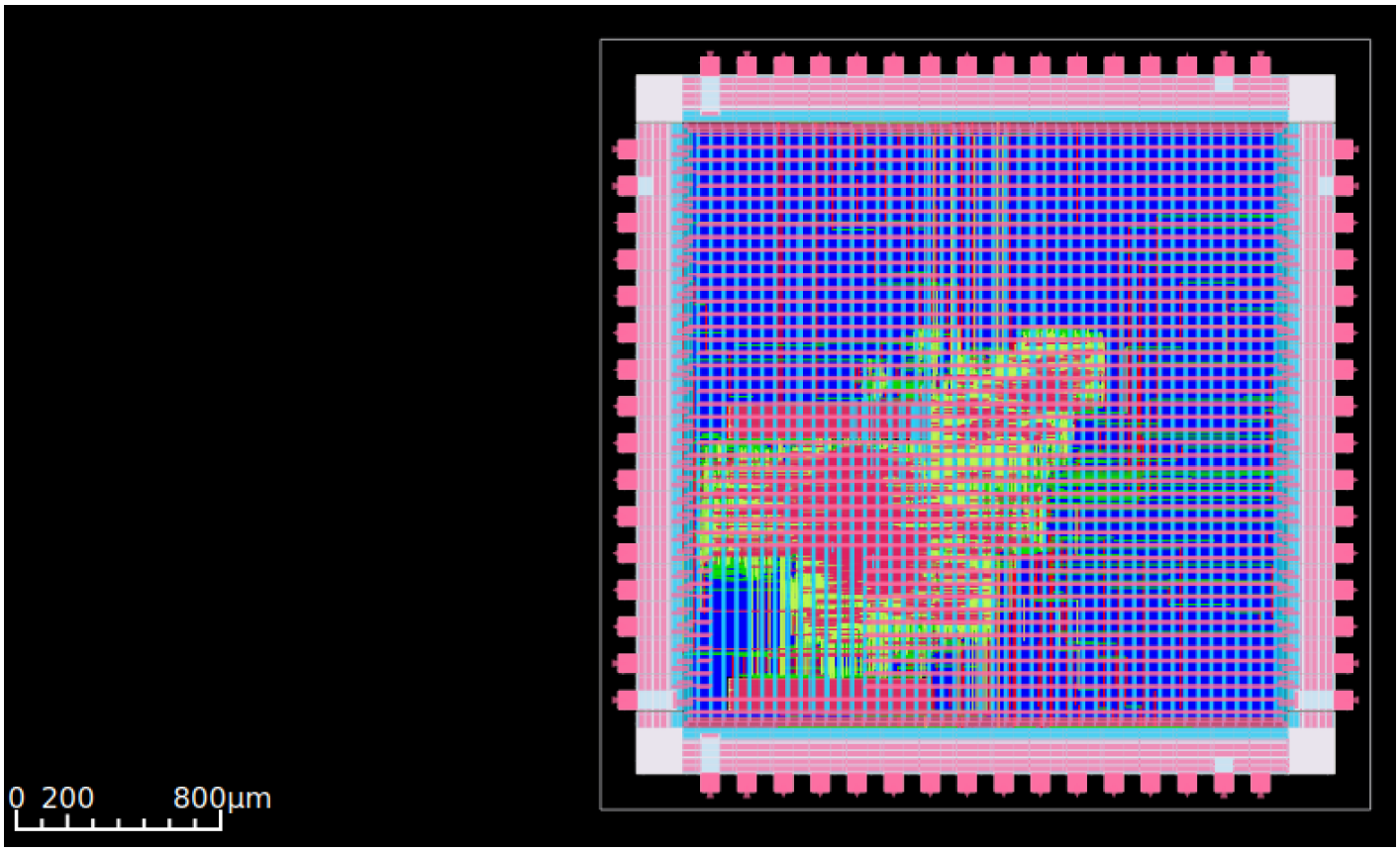

Abbildung 11: GDSII

Zu sehen ist, wie wenig Platz unsere Implementierung auf dem vorgesehenen 3mm - 3mm Chip einnimmt. Da dies nur der initelle Test für die Tapeout Pipeline war, zeigte uns dies, dass wir die Anzahl an Speicherzellen deutlich erhöhen können. In einem nächsten Schritt erweiterten wir die Anzahl an Speicherzellen von Standardmäßig 16 Zellen auf 1024 Zellen. Hierfür wurde lediglich die Konfigurationsdatei `cache_cfg_pkg.sv` angepasst.

10 Tests

Ausarbeitung Luca Schmid

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 verwendet 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`, welcher das Top-Level-Modul `redis_cache` instanziiert (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.

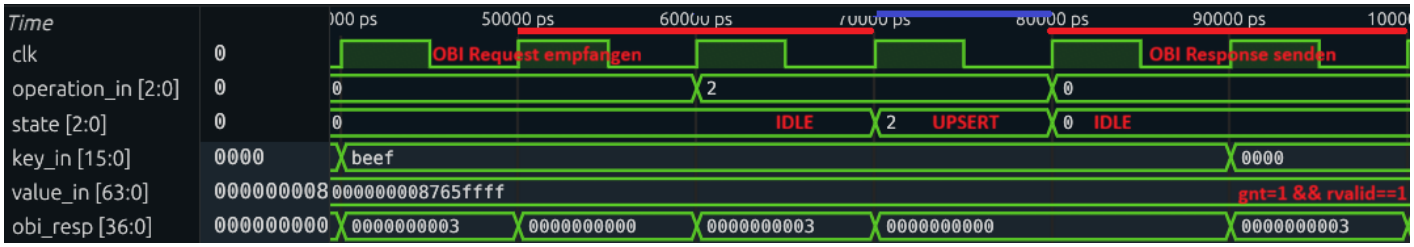

Abbildung 12: UPSERT (empty memory)

11 Learnings und Ausblick

Ausarbeitung gemeinsam

Nachfolgend werden unsere Learnings während der Entwicklung des Caches und der zugehörigen Toolchain beschrieben. Hierbei werden sowohl Learnings bezüglich der Implementierung als auch der generellen Arbeitsweise und des Projektmanagements beschrieben.

1) Hardware Implementierungs sind aus Sicht eines Softwareentwicklers deutlich umständlicher als Software. Hardware verhält sich anders als Software und muss anders angedacht werden.
2) Ein modularisiertes Design ermöglichte uns, die Arbeiten a) parallel zu bearbeiten und b) die Komplexität der einzelnen Module möglichst gering zu halten.
3) Selbst mit minimaler Anzahl an Featuren und mit der Idee der Erweiterung, hätten wir nicht gedacht, dass die Implementierung des Chips und die Integration in Croc so komplex sein würde.
4) Das Aufsetzen von Pipelines und Testautomatisierung über GitHub Actions ist für einen Projekt Scope von zwei Wochen zu aufwändig. Zuletzt wurden Tests als auch die Backendpipeline lokal ausgeführt.
5) Eine Optimierung auf der unterstenen Ebene (Taktzyklen) ermöglichte uns weitgehende Optimierungen der Statemaschinen vorzunehmen, was die Komplexität des gesamten Systems deutlich reduzierte.

Nachfolgend werden weitere Funktionalitäten beschrieben, welche wir gerne umgesetzt hätten, jedoch aufgrund von Zeitmangel nicht mehr implementieren konnten:

DELETE Implementation refactoren. Kann auf 1 State gekürzt werden Nach den Optimierungen des Memory Blockes könnte die DELETE Sub FSM auf einen einzigen State gekürzt werden. Hintergrund ist, dass bereits nach der positiven Flanke die DELETE operation abhängig vom HIT Signals des bereits schon anliegenden Schlüssels exisitert sodass nur noch abhängig von diesem Signal ein DELETE durchgeführt werden muss. Dieses Signal würde vom Memory Block bereits zur fallenden Taktflanke verarbeitet werden.

LIST Operation

Die zunächst angedachte Funktionalität einer LIST Operation wäre für eine realitätsnahe Umsetzung vermutlich sinnvoll. Damit müsste die darauf aufbauende Applikation nicht mehr intern die eigene Schlüsselverwaltung implementieren und diese direkt über die Hardware gestalten. Angedacht wäre, dass die LIST Operation über mehrere OBI Transaktionen die jegliche Schlüssel ausliest und an die Applikation zurückgibt.

TTL (Time To Live) Funktionalität

Redis implementiert eine TTL Funktionalität, welche Schlüssel nach einer vorgegebenen Zeit invalidiert und löscht. Ähnliches hatten wir für unsere Implementierung angedacht. Hierzu hätten wir einen Timer mit implementiert welcher immer den kleinsten TTL Wert der aktuell im Cache befindlichen Schlüssel verwendet und nach Ablauf der Zeit den Schlüssel automatisch löscht.

Evaluation des Nutzen von SRAM Unsere Aktuelle Implementierung verwendet für den Memory Block eine Liste von einzelnen selbst definierten Registerzellen. Eine Alternative hierzu wäre die Verwendung von bereits fertigen Memory Blöcken. Diese bieten den Vorteil, dass sie deutlich weniger Logik und Platz aufweisen, wobei die Ansteuerung und das Überprüfen der einzelnen Zellen auf einen *Hit* dadurch deutlich komplexer wird.