Grundlagenpraktikum: Rechnerarchitektur

Abschlussprojekt

Lehrstuhl für Design Automation

Organisatorisches

Auf den folgenden Seiten befindet sich die Aufgabenstellung zu eurem Projekt für das Praktikum. Die Rahmenbedingungen für die Bearbeitung werden in der Praktikumsordnung festgesetzt, die über Artemis[1] aufrufbar ist.

Ähnlich wie in den Hausaufgaben definiert die Aufgabenstellung ein zu implementierendes SystemC Modul. Dieses ist allerdings komplexer als in den Hausaufgaben und benötigt mehrere Untermodule für eine saubere Ausarbeitung. Besprecht deshalb innerhalb eurer Gruppe, welches Abstraktionsniveau für die Implementierung sinnvoll ist und diskutiert den Entwurf der Module gemeinsam.

Die Teile der Aufgabe, in denen C-Code anzufertigen ist, sind in C nach dem C17-Standard zu schreiben. Die Teile der Aufgabe, in denen C++-Code anzufertigen ist, sind in C++ nach dem C++14-Standard zu schreiben. Die jeweiligen Standardbibliotheken sind Teil der Sprachspezifikation und dürfen ebenfalls verwendet werden. Als SystemC-Version ist SystemC 2.3.3 oder 2.3.4 zu verwenden.

Die **Abgabe** erfolgt über das für eure Gruppe eingerichtete Projektrepository auf Artemis. Es werden keine Abgaben per E-Mail akzeptiert.

Die **Abschlusspräsentationen** finden nach der Abgabe statt. Die genauen Termine werden noch bekannt gegeben. Die Folien für die Präsentation müssen zur selben Deadline wie die Implementierung im Projektrepository im **PDF Format** abgegeben werden. Wie in der Praktikumsordnung besprochen sollen die Präsentationen eure Implementierung vorstellen und Ergebnisse der Literaturrecherche erklären. Außerdem sollte die Implementierung anhand **mindestens einer interessanten Metrik** (z.B. Anzahl an Gattern, I/O Analyse usw.) evaluiert und das Ergebnis dieser Evaluierung im Vortrag interpretiert und, wenn möglich, *mit Werten aus der Realität verglichen werden.* Zusätzlich sollte die Präsentation anhand einer Illustration kurz erklären, wie das implementierte Modul in die *TinyRISC* CPU aus den Hausaufgaben integriert werden könnte.

Zusätzlich zur Implementierung muss auch ein kurzer **Projektbericht** von bis zu 800 Wörtern im Markdown-Format abgegeben werden. Dieser sollte kurz angeben, welche Teile der Aufgabe von welchen Gruppenmitgliedern bearbeitet wurden und beschreiben, wie das implementierte Modul funktioniert. Außerdem sollte im Rahmen des Berichts eine kurze Literaturrecherche durchgeführt werden. Diese Literaturrecherche sollte sich auf das Thema eures Projekts konzentrieren und zumindest alle in der Einleitung **fett** gedruckten Begriffe erklären und die unten vorgeschlagenen Fragen beantworten. Quellenangaben für alle verwendeten Informationen sind willkommen und müssen nicht zum Wortlimit hinzugezählt werden.

Bei Fragen/Unklarheiten in Bezug auf den Ablauf und die Aufgabenstellung wendet euch bitte **schriftlich** über Zulip an euren Tutor.

Wir wünschen viel Erfolg und Freude bei der Bearbeitung der Aufgabe!

Mit freundlichen Grüßen
Die Praktikumsleitung

[1] `https://artemis.ase.in.tum.de/`

Ordnerstruktur

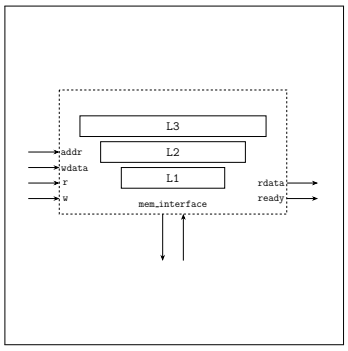
Die Abgabe muss ein `Makefile` im Wurzelverzeichnis enthalten, das über `make project` das Projekt kompilieren und die ausführbare Datei `project` erzeugen kann. Außerdem darf die Abgabe ein Shell-Script `build.sh` definieren, das den Build-Prozess startet. Dieses Build-Script wird von Artemis automatisch aufgerufen und seine Outputs werden als Testergebnis zurückgegeben. Damit kann kontrolliert werden, ob euer Projekt im Testsystem kompiliert und ausgeführt werden kann. Die Präsentationsfolien sollten unter dem Namen `slides.pdf` im Wurzelverzeichnis abgelegt werden.

Abgesehen von den oben genannten Punkten ist keine genaue Ordnerstruktur vorgeschrieben. Als Orientierung empfehlen wir aber die folgende Ordnerstruktur:

- `Makefile` — Das Makefile, das das Projekt kompiliert und die ausführbare Datei `project` erzeugt.
- `Readme.md` — Der Projektbericht im Markdown-Format.
- `build.sh` — Das Build-Script, das den Build-Prozess startet.
- `slides.pdf` — Die Folien der Abschlusspräsentation im PDF Format.
- `.gitignore` — Eine `.gitignore` Datei, die Verhindert, dass unerwünschte Dateien in das Git-Repository gelangen.
- `src/` — Ein Unterordner, der alle Quelldateien enthält.
- `include/` — Ein Unterordner, der alle Headerdateien enthält.
- `test/` — Ein Unterordner, der Dateien zum Testen (z.B. Test-Inputs) enthält.

Achtung: Kompilierte Dateien, IDE-spezifische Dateien, temporäre Dateien, Library-Code und überdurchschnittlich große Dateien sollten nicht im Repository enthalten sein. Diese Dateien können in der `.gitignore` Datei aufgelistet werden. Auf die SystemC Library kann, wie bei den Hausaufgaben, über die `SYSTEMC_HOME` Umgebungsvariable zugegriffen werden. Die SystemC Library muss und *darf* also nicht im Repository enthalten sein.

Wichtig: Das Makefile soll **eine** ausführbare Datei mit dem Namen `project` im aktuellen Ordner erstellen. Abweichungen von dieser Vorgabe können zu Abzügen führen.



Memory Cache

Speichert einen Teil des Hauptspeichers für schnellen Zugriff.

- ☐ Reduziert den Flaschenhals des Hauptspeichers.
- ☐ Mehrere Ebenen bieten verschiedene Geschwindigkeiten.
- ☐ Es werden verschiedene Zuordnungsstrategien verwendet.

Häufige Übertragung von Daten zwischen CPU und Hauptspeicher kann aufgrund der begrenzten Bandbreiten zu großen Zeitverlusten führen. Für Daten, die häufig benötigt werden, ist es effizienter, sie in einem schnelleren Speicher abzulegen.

Caches bieten diese Möglichkeit: In einem begrenzten Speicherbereich werden Daten aus dem Hauptspeicher zwischengespeichert, um den Zugriff zu beschleunigen.

Moderne CPUs verwenden oft mehrere von Caches; **Cache-Levels**. Bei Speicherzugriffen wird zuerst der L1 Cache überprüft, dieser ist meist am schnellsten aber auch am kleinsten. Werden die Daten dort nicht gefunden, wird in den nächsten Leveln (L2, L3, usw.) gesucht.

Wo Daten im Cache abgelegt werden, hängt von der **Mapping-Strategie** ab. Diese kann **direct-mapped** oder **fully associative** sein. Ist ein Cache direct-mapped, wird jeder Speicherblock nur in eine bestimmten Cache-Zeile abgelegt. Beim assoziativen Mapping kann ein Block in verschiedene Zeilen abgelegt werden - dafür muss aber eine "*Replacement-Strategie*" definiert werden, die entscheidet, welche Cache-Zeile ersetzt werden soll.

Caches sind oft in mehrere **Cachezeilen** unterteilt, die jeweils mehrere Bytes unabhängig voneinander speichern können.

SPEZIFIKATION: `CACHE`

Inputs

- ☐ `clk`: `bool (clock input)`
- ☐ `addr`: `uint32_t`
- ☐ `wdata`: `uint32_t`
- ☐ `r`: `bool`
- ☐ `w`: `bool`
- ☐ `mem_ready`: `bool`
- ☐ `mem_rdata`: `uint32_t`

Outputs

- ☐ `rdata`: `uint32_t`
- ☐ `ready`: `bool`
- ☐ `miss`: `bool`
- ☐ `mem_addr`: `uint32_t`
- ☐ `mem_wdata`: `uint32_t`
- ☐ `mem_r`: `bool`
- ☐ `mem_w`: `bool`

Implementation
Die Inputs `r` und `w` bestimmen, ob ein Lese- oder Schreibzugriff durchgeführt werden soll. Ist einer der beiden Inputs gesetzt, wird der Cache beim nächsten Clock-Zyklus gestartet. Dafür wird der Output `ready` zuerst auf 0 gesetzt.

Bei Lesezugriffen wird die Adresse `addr` dann im Cache gesucht, beginnend mit der ersten Cache-Ebene. Wird sie in einer der Cache-Ebenen gefunden, wird der darin liegende Wert in `rdata` gespeichert und `ready` auf 1 gesetzt. Dieser Vorgang soll `--latency-cache-X*` Clockzyklen (mit `X` = die jeweilige Cache-Ebene) benötigen. Wenn die Adresse in keiner Cache-Ebene gefunden wird, findet ein Cache Miss statt. (Lesezugriffe können auf allen Cache-Ebenen gleichzeitig gestartet werden. Wenn sich der gesuchte Wert z.B. auf Ebene 2 befindet, muss deshalb nur die Latenz für Ebene 2 verrechnet werden, nicht auch noch für Ebene 1)

Der Cache soll auch (write-through) Schreibzugriffe unterstützen. Wird ein Schreibzugriff durchgeführt, wird die Adresse `addr` in den verschiedenen Cache-Ebenen gesucht. Wird sie gefunden, wird `wdata` mit *Little-Endian*-Encoding in die Ebene geschrieben. Außerdem muss der Wert dann auch in allen höheren Cache-Ebenen aktualisiert und in den Hauptspeicher geschrieben werden. Dieser Vorgang wird durch die Latenz des Schreibzugriffs auf den Hauptspeicher dominiert, die Latenz zum Schreiben auf die einzelnen Cache-Ebenen kann daher vernachlässigt werden. Das Schreiben auf den Hauptspeicher funktioniert Analog zum Lesenzugriff, wie in Cache Miss beschrieben.
Wenn die Adresse in keiner Cache-Ebene gefunden wird, findet zuerst ein Cache Miss statt, bevor der Schreibzugriff durchgeführt wird.

Cache Miss
Ein Cache Miss tritt auf, wenn die Adresse in keiner Cache-Ebene gefunden wird. In diesem Fall muss der Wert aus dem Hauptspeicher gelesen werden. Für die Kommunikation mit dem Hauptspeicher werden die `mem_` Inputs und Outputs verwendet. Der Cache soll dafür die Outputs `mem_r` und `mem_addr` entsprechend setzen und dann warten, bis `mem_ready` auf 1 gesetzt wird. Danach kann der Wert aus `mem_rdata` gelesen werden.
Der gelesene Wert muss dann im Cache abgelegt werden. Jede Cache-Ebene besteht aus mehreren Cachezeilen, gegeben durch `--num-lines-X*`. Eine Cachezeile kann jeweils eine begrenzte Menge an Daten speichern. Die genaue Zeilengröße wird durch `--cacheline-size*` angegeben. Die Cachezeile, in die der gelesene Wert abgelegt wird, hängt von der *Mapping-Strategie* ab, die durch `--mapping-strategy*` bestimmt wird.
Bei jedem Cache Miss muss der Output `miss` auf 1 gesetzt werden. Zu Beginn der nächsten Cache-Operation muss er dann aber wieder zurückgesetzt werden.
Wir gehen davon aus, dass hier der Zeitaufwand für den Speicherzugriff dominiert. Es muss daher nicht auch noch für das Schreiben in den Cache gewartet werden.

Methoden
Das `CACHE` Modul stellt außerdem folgende Methoden zur Verfügung:

- ☐ `uint8_t getCacheLineContent(uint32_t level, uint32_t lineIndex, uint32_t index)`:

  Gibt das Byte zurück, das in Cache-Level ***`level`****, Cachezeile* ***`lineIndex`*** *an der Stelle* ***`index`*** *steht.*

Alle Methoden, die in diesem Absatz beschrieben wurden, dürfen mit beliebig viel *Magie* implementiert werden.

Erlaubte *Magie*: Rechenoperationen, Speichern von Werten und Flags, Ersetzungsstrategien, Warten auf Signale, Suche von Werten im Cache, sammeln von Statistiken (hits, misses, etc.).

Optionen*

Die folgende Liste zählt alle zusätzlichen Optionen auf, die vom Rahmenprogramm erkannt und angewendet werden müssen. Für jede dieser Optionen soll der Konstruktor des Hauptmoduls außerdem in dieser Reihenfolge *einen entsprechenden Parameter annehmen, der den Wert für das Modul setzt.*

- ☐ `--num-cache-levels`: `uint8_t` — *Anzahl an Cache-Levels (min. 1, max. 3)*
- ☐ `--cacheline-size`: `uint32_t` — *Die Größe einer Cachezeile in Bytes*
- ☐ `--num-lines-l1`: `uint32_t` — *Die Anzahl an Cachezeilen im L1-Cache*
- ☐ `--num-lines-l2`: `uint32_t` — *Die Anzahl an Cachezeilen im L2-Cache*
- ☐ `--num-lines-l3`: `uint32_t` — *Die Anzahl an Cachezeilen im L3-Cache*
- ☐ `--latency-cache-l1`: `uint32_t` — *Latenz des L1-Caches in Clockzyklen*
- ☐ `--latency-cache-l2`: `uint32_t` — *Latenz des L2-Caches in Clockzyklen*
- ☐ `--latency-cache-l3`: `uint32_t` — *Latenz des L3-Caches in Clockzyklen*
- ☐ `--mapping-strategy`: `uint8_t` — *Die gewünschte Mapping-Strategie: 0 =* direct-mapped*, 1 =* fully associative

Weitere Hinweise

- ☐ Um den Cache testen zu können wird auch ein Hauptspeicher benötigt. Dafür kann das `MAIN_MEMORY` Modul aus den Hausaufgaben verwendet, oder ein neues Speichermodul erstellt werden.
- ☐ Der Cache soll inklusiv sein. Das bedeutet, dass alle Daten im L1-Cache auch im L2-Cache liegen müssen.

- ☐ Die Replacement-Strategie (LRU, LFU...) für den Cache ist nicht spezifiziert. Sie kann selbst gewählt werden und sollte beim Vortrag kurz erläutert werden.
- ☐ Die `getCacheLineContent` Methode soll nur für Tests verwendet werden und nicht in der eigentlichen Implementierung.
- ☐ Das Cache Modul muss nur Zweierpotenzen als Cachezeilengröße unterstützen.
- ☐ Das gesamte Cache Modul kann und soll aus mehreren Untermodulen (z.B. für die einzelnen Cache-Level) bestehen.
- ☐ Der Speicher soll Byte-Adressierbar sein: Auf jede Adresse kann zugregriffen werden. Es werden immer 4 Bytes, beginnend bei der angegebenen Adresse, gelesen oder geschrieben.

Fragen für die Literaturrecherche
Zusätzlich zu den in der Einleitung markierten Fachbegriffen, sollte die Literaturrecherche auch folgende Fragen beantworten:

- ☐ Was sind typische Größen für Caches, Cachezeilen und Latenzen in modernen CPUs?
- ☐ Welche Replacement Strategies gibt es und was sind ihre Vor- und Nachteile?
- ☐ Untersucht das Speicherzugriffsverhalten eines speicherintensiven Algorithmus. Erstellt dann csv-Dateien, die beispielhaft die Speicherzugriffe des Algorithmus beschreiben.

Rahmenprogramm

Ein Rahmenprogramm soll in C implementiert werden, über das das Modul getestet werden kann. Das Rahmenprogramm soll in der Lage sein, verschiedene CLI Optionen einzulesen und das Modul entsprechend zu konfigurieren. Für jede der Optionen sollte ein sinnvoller Standardwert festgelegt werden. Zusätzlich zu den oben genannten Modul-spezifischen Optionen soll das Rahmenprogramm folgende CLI Parameter unterstützen:

- ☐ `--cycles`: `uint32_t` — *Die Anzahl der Zyklen, die simuliert werden sollen.*
- ☐ `--tf`: `string` — *Der Pfad zum Tracefile. Wenn diese Option nicht gesetzt wird, soll kein Tracefile erstellt werden.*
- ☐ `<file>`: `string` — *Positional Argument: Der Pfad zur Eingabedatei, die verwendet werden soll.*
- ☐ `--help`: `flag` — *Gibt eine Beschreibung aller Optionen des Programms aus und beendet die Ausführung.*

Ein einfacher Aufruf des Programms könnte dann so aussehen:

```
./project --cycles 1000 requests.csv
```

Es dürfen zum Testen auch weitere Optionen implementeirt werden, das Programm muss aber auch mit nur den oben genannten Optionen ausführbar sein.

Für jede Option muss getestet werden, ob die Eingabe gültig ist (reichen Zugriffsrechte auf Dateien aus, sind die Werte in einem gültigen Bereich, etc.). Wenn ein Wert falsch übergeben wird, soll eine sinnvolle Fehlermeldung ausgegeben werden und das Programm beendet werden.

Bei Verwendung der Option `--tf` soll ein Tracefile erstellt werden. Das Tracefile soll die wichtigsten verwendeten Signale beinhalten.

Zum Einlesen der CLI Parameter empfehlen wir `getopt_long` zu verwenden. `getopt_long` akzeptiert auch Optionen, die nicht zur Gänze ausgeschrieben sind. Dieses Feature steht zur Verwendung frei, muss aber nicht verwendet werden.

Alle übergebenen Optionen sollen im Rahmenprogramm verarbeitet werden. In C++ sollte dann die folgende Funktion implementiert werden:

```
struct Result run_simulation(
    uint32_t cycles,
    const char* tracefile,
    uint8_t numCacheLevels,
    uint32_t cachelineSize,
    uint32_t numLinesL1,
    uint32_t numLinesL2,
    uint32_t numLinesL3,
    uint32_t latencyCacheL1,
    uint32_t latencyCacheL2,
```

```
    uint32_t latencyCacheL3,
    uint8_t mappingStrategy,
    uint32_t numRequests,
    struct Request* requests,
);
```

In dieser Funktion wird das `CACHE` Modul initialisiert und die Simulation gestartet. Die Ergebnisse der Simulation sollen in einem `Result` Struct zurückgegeben werden.

```
struct Result {
    uint32_t cycles;
    uint32_t misses;
    uint32_t hits;
};
```

Dieses Struct beinhaltet Statistiken der Simulation, wie die Anzahl der zur Abarbeitung benötigen Zyklen und die Anzahl der Cache Misses und Hits. Alle wichtigen Informationen des `Result` Structs sollten nach der Ausführung in der Kommandozeile anschaulich ausgegeben werden.

Der Parameter `requests` beinhaltet eine Liste von `numRequests` `Request` Structs:

```
struct Request {
    uint32_t addr;
    uint32_t data;
    uint8_t w;
};
```

Diese Requests sollten durch den Cache nacheinander abgearbeitet werden. Wenn `w 1` ist, soll ein Schreibzugriff mit dem Wert in `data` an Adresse `addr` durchgeführt werden. Ansonsten soll der Wert an der jeweiligen Adresse gelesen und in `data` gespeichert werden.

Innerhalb von `run_simulation` kann davon ausgegangen werden, dass alle übergebenen `Request` Structs gültig sind.

Eingabedatei

Die Eingabedatei beinhaltet eine Liste von Anfragen, die während der Simulation abgearbeitet werden sollen. Sie ist als *csv*-Datei formatiert und sollte noch im Rahmenprogramm eingelesen und zu `Request` Structs verarbeitet werden. Sie hat folgendes Format:

Type	Address	Data
W	0x0010	20
R	0x0010	
W	123	0x12
⋮	⋮	⋮

Die *csv*-Datei **muss** *mit Header-Zeile* und dem Separator `","` eingelesen werden. Zusätzliche Features, z.B. zur automatischen Erkennung des Separators oder Unterstützung von Eingabedateien ohne Header-Zeile sind erlaubt aber nicht benötigt.

Die erste Spalte unterscheidet zwischen Lesezugriffen (R) und Schreibzugriffen (W). Die zweite Spalte gibt die Adresse an, auf die zugegriffen werden soll. Die dritte Spalte gibt den Wert an, der geschrieben werden soll. Bei Lesezugriffen muss dieser Wert immer leer sein. Adressen und Werte können sowohl im Dezimal- als auch im Hexadezimalformat angegeben werden.

Jegliche Fehler in der Eingabedatei sollen als Fehlermeldung ausgegeben werden und das Programm beenden.