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

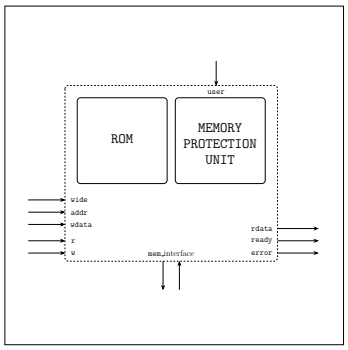
Die Abgabe muss ein `Makefile` im Wurzelverzeichnis enthalten, das über `make project` das Projekt kompilieren und die ausführbare Datei `project` erzeugen kann. Außerdem darf die Abgabe ein Shell-Script `build.sh` definieren, das den Build-Prozess startet. Dieses Build-Script wird von Artemis automatisch 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 Controller

Verwaltet Zugriffe auf den Speicher.

- ☐ Kontrolliert Zugriffe auf den Hauptspeicher.
- ☐ Ermöglicht effizientes Memory Mapping.
- ☐ Erlaubt Speicherzugriffe mit verschiedenen Datenbreiten.

Das Speichern und Lesen von Daten aus dem Hauptspeicher ist eine der wichtigsten Aufgaben eines Prozessors. Gleichzeitig birgt Speicherzugriff aber auch viele Gefahren und Herausforderungen, vor allem in **von Neumann Architekturen**.

Memory Controller werden deshalb verwendet, um Speicherzugriffe zu koordinieren und zu optimieren. Sie bieten deshalb eine große Menge an Funktionalitäten, um Speicherzugriffe zu beschleunigen und zu vereinfachen. Die tatsächlich unterstützen Funktionalitäten hängen dabei aber stark von der konkreten Implementierung ab. In dieser Aufgabe implementieren wir drei Funktionalitäten. Je nach Hersteller sind diese oft eigene Module oder Teile bereits existierender Module. Für diese Aufgabe fassen wir sie allerdings als ein Memory Controller Modul zusammen:

- ☐ **Memory Mapping:** Ermöglicht es, Speicheradressen auf verschiedene Speicherbereiche abzubilden. Somit kann z.B. auf **ROM** oder verschiedene Peripheriegeräte zugegriffen werden.
- ☐ **Memory Protection Unit** Verhindert den Zugriff auf Speicherbereiche, die für ein Programm nicht vorgesehen sind.
- ☐ Data-Width Adaptation: Ermöglicht es, Speicherzugriffe mit verschiedenen Datenbreiten durchzuführen.

SPEZIFIKATION: `MEMORY_CONTROLLER`

Inputs	*Outputs*
☐ `clk`: `bool (clock input)`	☐ `rdata`: `uint32_t`
☐ `addr`: `uint32_t`	☐ `ready`: `bool`
☐ `wdata`: `uint32_t`	☐ `error`: `bool`
☐ `r`: `bool`	
☐ `w`: `bool`	
☐ `wide`: `bool`	
☐ `user`: `uint8_t`	☐ `mem_addr`: `uint32_t`
	☐ `mem_wdata`: `uint32_t`
☐ `mem_ready`: `bool`	☐ `mem_r`: `bool`
☐ `mem_rdata`: `uint32_t`	☐ `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.

Der Memory Controller beinhaltet eine *Read-Only-Memory* Einheit, die Speicher mit dynamischer Größe zur Verfügung stellt. Der Inhalt des ROMs wird über den Konstruktor des Moduls festgelegt und kann nicht verändert werden. Die Speicheradressen `0x0` (inklusiv) bis `--rom-size*` (exklusiv) werden auf den ROM-Bereich abgebildet. Findet ein Lesezugriff auf eine Adresse (gegeben durch den Input `addr`) innerhalb dieses Bereichs statt, wird der Wert aus dem ROM an der jeweiligen Position auf den Output `rdata` geschrieben und `ready` wird auf `1` gesetzt. Findet ein Schreibzugriff auf eine Adresse im ROM statt, führt das zu einem Zugriffsfehler. Das Lesen aus dem ROM benötigt `--latency-rom*` Clockzyklen.

Speicherzugriffe außerhalb des ROM Bereichs werden an den Hauptspeicher weitergeleitet. Zuerst muss allerdings eine Kontrolle der Zugriffsberechtigung durchgeführt werden. Dafür wird überprüft, ob der angegebene Benutzer `user` Zugriff auf die gewünschte Adresse hat. Anfangs darfs jeder Benutzer auf jede Adresse zugreifen. Sobald ein Schreibzugriff auf eine Adresse stattgefunden hat, darf nur noch der schreibende Benutzer auf den dazugehörigen Speicherblock zugreifen. Users `0` und `255` haben immer Zugriff auf alle Adressen. Außerdem gibt ein Zugriff von User `255` auf eine Adresse den zugehörigen Hardwareblock wieder frei und erlaubt somit wieder Zugriffe von allen Benutzern. Beim nächsten Schreibzugriff eines Nutzers wird der Bereicht allerdings wieder diesem Nutzer zugeschrieben. Für einen erfolgreichen Speicherzugriff muss der Benutzer die Berechtigung für jedes Byte besitzen, auf das zugegriffen wird. Zugriffe ohne Berechtigung führen zu einem Zugriffsfehler. Dieser Vorgang wird durch die Latenz des Zugriffs auf den Hauptspeicher dominiert, die Latenz für die Zugriffsüberprüfung ist vernachlässigbar.

Für den Zugriff auf den Hauptspeicher werden die `mem_` Inputs und Outputs verwendet. Die jeweiligen Outputs werden dafür entsprechend gesetzt und der Memory Controller wartet auf das `mem_ready` Signal.

Bei *allen* Speicherzugriffen soll *Data-Width-Adaptation* durchgeführt werden: Der Memory Controller ist in der Lage, Reads und Writes mit einer Bandbreite von 4 Bytes oder 1 Byte durchzuführen. Der Input `wide` gibt dafür an, ob ein 4-Byte Zugriff durchgeführt werden soll. Der Speicherbereich soll aber unabhängig von der Bandbreite Byte-Adressierbar sein, also sollten Zugriffe auf alle Adressen unterstützt werden. Der Hauptspeicher unterstützt nur 4-Byte Zugriffe. Bei einem 1-Byte Lesezugriff muss der Memory Controller nach dem Lesen das *Least-Significant-Byte* aus dem gelesenen Wert bestimmen und auf den Output `rdata` schreiben. Die restlichen Bytes von `rdata` sollen auf `0` gesetzt werden.
Bei einem 1-Byte Schreibzugriff muss zuerst der Wert an der gewünschten Adresse im Hauptspeicher gelesen werden, um damit die restlichen Bytes von `mem_wdata` zu befüllen damit keine Daten verloren gehen.

Nach dem Abschluss jeses Speicherzugriffs wird der Output `ready` auf `1` gesetzt.

Zugriffsfehler

Bei einem Zugriffsfehler wird der Speicherzugriff sofort abgebrochen. In diesem Fall werden `ready` und `error` auf `1` gesetzt. Bei Speicherzugriffen, die keinen Zugriffsfehler auslösen, muss `error` wieder auf `0` gesetzt werden.

Speicherblöcke

Der Hauptspeicher wird in mehrere Blöcke aufgeteilt. Jeder Block hat eine Größe von `--block-size*` Bytes. Bei einer Blockgröße von `0x1000` verläuft der erste Speicherblock (nach einem ROM-Bereich mit `--rom-size*=0x100000`) beispielsweise von `0x100000` bis `0x101000`, der zweite von `0x101000` bis `0x102000` usw.

Methoden

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

- ☐ `uint8_t getOwner(uint32_t address)`:

  Gibt den User an, der Zugriff auf den Block hat, der die angegebene Adresse beinhaltet. Wenn die Adresse keinem User zugewiesen ist, gibt diese Methode `255` zurück.

- ☐ `void setRomAt(uint32_t address, uint8_t data)`:

  Setzt den Wert im ROM an der angegebenen Adresse auf den gegebenen Wert.

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

Erlaubte *Magie*: Rechenoperationen, Speichern von Werten und Flags, Kontrolle von Zugriffsrechten, Warten auf Signale, Data-Width-Adaptation, sammeln von Statistiken.

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.*

- ☐ `--latency-rom`: `uint32_t` — *Latenz des ROMs in Clockzyklen*
- ☐ `--rom-size`: `uint32_t` — *Größe des ROMs in Bytes (min. 0)*
- ☐ `--block-size`: `uint32_t` — *Die Größe eines Speicherblocks in Bytes*
- ☐ `--rom-content`: `const char*` — *Der Pfad zu einer Datei, die den Inhalt des ROMs enthält. Die Datei besteht aus einer Liste von 32-Bit Werten im Dezimal- oder Hexadezimalformat, die durch Zeilenumbrüche geteilt sind. Sie soll im Rahmenprogramm eingelesen werden. Der Konstruktor des `MEMORY_CONTROLLER` Moduls soll stattdessen den Parameter `uint32_t* romContent` annehmen, der die bereits eingelesenen Werte beinhaltet.*

Weitere Hinweise

- ☐ Um den Memory Controller 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 ROM-Bereich muss nur Zweierpotenzen als Größen unterstützen.
- ☐ Das gesamte Memory Controller Modul kann und soll weitere Untermodule (z.B. für den ROM Speicher) verwenden.
- ☐ Die ROM-Datei darf auch weniger Werte enthalten, als die Größe des ROMs angibt. In diesem Fall sollen die restlichen Werte auf `0` gesetzt werden.
- ☐ Wenn die ROM-Datei mehr Werte beinhaltet, als die Größe des ROMs angibt, wird das als Fehler angesehen und das Programm soll sofort beendet werden.
- ☐ Der Parameter `romContent` im Konstruktor des `MEMORY_CONTROLLER` Moduls soll Zugriff auf die Werte im ROM bieten. Der `MEMORY_CONTROLLER` (oder ein mögliches `ROM` Untermodul) darf diese Werte in eine eigene Datenstruktur kopieren oder den Pointer selbst abspeichern und verwenden.

Fragen für die Literaturrecherche

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

- ☐ Wie unterscheiden sich die *von Neumann* und *Harvard* Architetkuren?
- ☐ In welchen Situationen wird Memory Mapping verwendet?

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,
    uint32_t latencyRom,
    uint32_t romSize,
    uint32_t blockSize,
    uint32_t* romContent,
    uint32_t numRequests,
    struct Request* requests,
);
```

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

```
struct Result {
    uint32_t cycles;
    uint32_t errors;
};
```

Dieses Struct beinhaltet Statistiken der Simulation, wie die Anzahl der zur Abarbeitung benötigen Zyklen und die Anzahl der Zugriffsfehler. 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;
    uint8_t user;
    uint8_t wide;
};
```

Diese requests sollten durch den Memory Controller 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. `user` und `wide` geben jeweils die dazugehörigen Werte der Inputs `user` und `wide` des Memory Controllers an.

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	User	Wide
W	0x0010	20	0	F
R	0x0010		5	T
W	123	0x12	0xff	F
⋮	⋮	⋮	⋮	⋮

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. Spalte 4 gibt den User für den jeweiligen Speicherzugriff an. User, Adressen und

Werte können sowohl im Dezimal- als auch im Hexadezimalformat angegeben werden. Spalten 5 gibt an, ob ein 4-Byte Zugriff durchgeführt werden soll (T) oder nicht (F). Bei 1-Byte Schreibzugriffen darf der zu schreibende Wert nicht mehr als 1 Byte Speicher einnehmen. Ansonsten gilt die Eingabedatei als fehlerhaft.

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