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

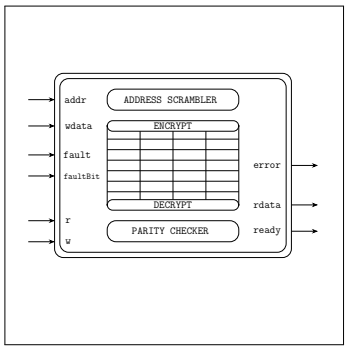
Die Abgabe muss ein `Makefile` im Wurzelverzeichnis enthalten, das über `make project` das Projekt kompilieren und die ausführbare Datei `project` erzeugen kann. Außerdem darf die Abgabe ein Shell-Script `build.sh` definieren, das den Build-Prozess startet. Dieses Build-Script wird von Artemis automatisch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



Secure-Memory Unit

Sammelt verschiedene Security Features um sensitive Daten zu handhaben.

- ☐ Speichert Daten verschlüsselt ab.
- ☐ Verwendet Address Scrambling um Angriffe abzuwehren.
- ☐ Prüft die Korrektheit von gespeicherten Daten.

Mit steigenden Anforderungen an Sicherheit und immer mehr Fällen an Datenmissbrauch ist es wichtig, sicher mit sensitiven Daten umzugehen. Verschiedene Sicherheitsfeatures können dabei helfen, die Daten zu schützen. In diesem Projekt soll ein Modul entwickelt werden, das einige dieser Sicherheitsfeatures zu einem Secure-Memory Unit Modul zusammenfasst[1]. Das Modul stellt einen Speicherbereich bereit, der mit verschiedenen Algorithmen von Angriffen und Fehlern geschützt wird.

Alle Daten, die in die Secure-Memory Unit geschrieben werden, werden verschlüsselt abgespeichert. Dafür wird ein **symmetrischer Verschlüsselungsalgorithmus** verwendet: Ein 8-Bit Schlüssel wird automatisch generiert und mittels **XOR Verschlüsselung** auf jedes geschriebene Byte angewandt.

Außerdem unterstützt die Secure-Memory Unit **Address Scrambling**: Mit einem 32-Bit Schlüssel wird jede Adresse, auf die zugegriffen wird, zu einer anderen physischen Adresse umgewandelt. Alle benötigten Schlüssel werden automatisch von einem **Pseudo-Random Number Generator** erstellt.

Die Secure-Memory Unit geht davon aus, dass der Speicher fehlerhaft sein kann. Sie verwendet deshalb **Parity-Bits**, um die Korrektheit der gespeicherten Daten zu überprüfen. Um Fehler im Speicher zu simulieren bietet das Modul außerdem eine Methode, um zufällige Bits in den gespeicherten Daten zu invertieren.

SPEZIFIKATION: `SECURE_MEMORY_UNIT`

Inputs

- ☐ `clk`: `bool (clock input)`
- ☐ `addr`: `uint32_t`
- ☐ `wdata`: `uint32_t`
- ☐ `r`: `bool`
- ☐ `w`: `bool`
- ☐ `fault`: `uint32_t`
- ☐ `faultBit`: `sc_bv<4>`

Outputs

- ☐ `rdata`: `uint32_t`
- ☐ `ready`: `bool`
- ☐ `error`: `bool`

[1] Moderne CPUs haben gewöhnlich keine eigene "Secure-Memory Unit", sondern verwenden diese Algorithmen direkt in den Komponenten die sie konkret benötigen. Diese "Secure-Memory Unit" hilft nur, die verschiedenen Security-Konzepte bündig auszuarbeiten.

Implementation
Wenn zur steigenden Flanke von `clk` `r` oder `w` auf `1` gesetzt ist, soll ein Lese- oder Schreibzugriff gestartet werden. Dafür wird zuerst der Output `ready` auf `0` gesetzt.

Das Modul beginnt mit der Berechnung der Adresssen. Bei Schreibzugriffen wird der Wert aus `wdata` in 4 Teile aus je 8 Bits aufgeteilt. `--endianness*` bestimmt, ob sie als Little-Endian oder Big-Endian abgespeichert werden sollen. Jeder dieser Teile wird dann mit einem 8-Bit Schlüssel mittels XOR Verschlüsselung verschlüsselt. Dieser Schlüssel sollte zu Beginn der Simulation zufällig erstellt werden. Da das Verschlüsseln des Inputs und das Berechnen der Adresse parallel durchgeführt werden können, wird die Anzahl der Zyklen, die für diesen Schritt nötig sind, durch das Maximum von `--latency-scrambling*` und `--latency-encrypt*` bestimmt.

Die 4 Adressen für die Sub-Bytes des Wertes, auf den Zugegriffen wird und die während der Berechnung der Adressen bestimmt wurden werden dann verwendet, um Byte für Byte Werte aus dem Speicher zu lesen oder in den Speicher zu schreiben. Dieser Vorgang benötigt für alle Sub-Bytes *insgesamt* `--latency-memory-access*` Clockzyklen zur Durchführung.

Nach einem Schreibzugriff wird für jedes der geschriebenen Bytes ein *Parity Bit* berechnet. Dafür wird gezählt, wie oft die Binärdarstellung des jeweiligen Bytes die Ziffer `1` enthält. Ist diese Anzahl gerade, dann ist das Parity Bit `0`, ansonsten ist es `1`. Für jedes gespeicherte Byte wird dieses Parity Bit dann im Modul abgespeichert. Dieser Vorgang benötigt keine zusätzlichen Clockzyklen und wird sofort abgeschlossen.

Nach einem Lesezugriff müssen die vier Bytes wieder mit demselben Encryption-Key entschlüsselt und basierend auf `--endianness` wieder zu einem 4-Byte Word zusammengefügt werden. Außerdem muss für jedes gelesene Byte ein Parity Check durchgeführt werden: Die Anzahl an `1` in der Binärdarstellung der Zahl muss dafür berechnet werden und es wird überprüft, ob das Parity Bit, das für das jeweilige Byte gespeichert wurde, noch damit übereinstimmt. Ist das nicht der Fall, so wird `error` auf `1` gesetzt und `0` wird in `rdata` geschrieben. Ansonsten wird das gelesene Word in `rdata` geschrieben und `error` wird auf `0` gesetzt. Da hier Address Scrambling und Entschlüsselung nicht gleichzeitig durchgeführt werden können, benötigt dieser Vorgang `--latency-encrypt*` + `--latency-scrambling*` Clockzyklen.

Unabhängig von der durchgeführten Operation wird nach Abschluss jedes Speicherzugriffs `ready` auf `1` gesetzt. Wenn beim Speicherzugriff kein Fehler aufgetreten ist muss `error` auf `0` gesetzt werden.

Address Scrambling

Adressen bestehen aus 32 Bits und weißen immer auf einen Speicherort für ein 4-Byte Word hin. Mit *Address Scrambling* werden alternative physische Adressen für gewünschte Speicheradressen berechnet. Dafür werden aus der gewünschten Adresse zuerst 4 Unteradressen für die einzelnen Bytes berechnet. Aus der Adresse `0x10010000` entstehen so zum Beispiel die Unteradressen `0x10010000`, `0x10010001`, `0x10010002` und `0x10010003`.

Nun wird ein 32-Bit Schlüssel mittels XOR Verschlüsselung auf jede dieser Adressen angewandt. Damit entstehen 4 physiche Unteradressen für die gewünschte Adresse. Der Schlüssel sollte zu Beginn der Simulation zufällig erstellt worden sein.

Pseudo-Random Number Generator (pRNG)

Das `SECURE_MEMORY_UNIT` Modul soll in der Lage sein, selbst zufällige Zahlen zu generieren. Dafür gibt es verschiedene Algorithmen, wir empfehlen aber die Verwendung eines *linearen Kongruenzgenerators*[2]

Die genaue Implementierung des pRNGs steht euch frei. Es muss allerdings ein Algorithmus verwendet werden, dessen Startwert mit einem *Seed* bestimmt werden kann und je nach Seed unterschiedliche Werte generiert. Der Seed wird durch `--seed*` bestimmt.

Fault Injection

Um das Parity-Checking zu testen, bietet das `SECURE_MEMORY_UNIT` Modul auch eine Möglichkeit zur *Fault Injection*. Zu *jedem* Clockzyklus wird der Wert in `fault` überprüft. Wenn der Wert `UINT32_MAX` entspricht, passiert nichts. Ansonsten wird ein Bit in der physischen Adresse `fault` invertiert. Das invertierte Bit hängt vom Wert in `faultBit` ab. Bei `0` wird das Least-Significant-Bit invertiert, bei `7` das Most-Significant-Bit. Bei einem Wert von `faultBit = 8` wird das Parity-Bit, das zur jeweiligen physischen Adresse gehört, invertiert.

Methoden

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

- ☐ `void setScramblingKey(uint32_t key)`:

  Setzt den Schlüssel, der zum Address Scrambling verwendet wird.

- ☐ `void setEncryptionKey(uint32_t key)`:

  Setzt den Schlüssel, der zum Verschlüsseln/Entschlüsseln verwendet wird.

- ☐ `uint8_t getByteAt(uint32_t physicalAddress)`:

  Gibt das Byte an der gegebenen physischen Adresse zurück.

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

Erlaubte *Magie*: Rechenoperationen, Verschlüsselung/Entschlüsselung, Parity-Bit Berechnung und Check, Berechnung der Adressen, Fault Injection, Speichern von Werten und Flags.

[2] Auch wenn es in dieser Aufgabe um eine *sichere* Speichereinheit geht, weißen wir darauf hin, dass diese Art von pRNG *nicht* kryptographisch sicher ist.

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.*

- ☐ `--endianness`: `uint8_t` — *Gibt an, ob Little-Endian (0) oder Big-Endian (1) verwendet werden soll.*
- ☐ `--latency-scrambling`: `uint32_t` — *Die Latenz für Address Scrambling in Clockzyklen.*
- ☐ `--latency-encrypt`: `uint32_t` — *Die Latenz für Verschlüsselung/Entschlüsselung in Clockzyklen.*
- ☐ `--latency-memory-access`: `uint32_t` — *Die Latenz für Speicherzugriffe in Clockzyklen.*
- ☐ `--seed`: `uint32_t` — *Der Seed, der für den pRNG verwendet werden soll.*

Weitere Hinweise

- ☐ Es soll *erlaubt* sein, aus Adressen zu lesen, auf die noch nicht geschrieben wurde, ohne einen Fehler im Parity-Check auszulösen (solange keine Fault Injection darauf angewendet wurde).
- ☐ Bei Schreibzugriffen kann kein Parity Fehler passieren. `error` muss daher nach jedem Schreibzugriff auf `0` gesetzt werden.
- ☐ Fault Injection muss zu *JEDEM* Clockzyklus möglich sein.
- ☐ Wenn eine Fault Injection während eines Speicherzugriffs auftritt, kann sie nach belieben *sinnvoll* gehandhabt werden.

Fragen für die Literaturrecherche

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

- ☐ Kann mit dem Parity Check *jeder* Fehler gefunden werden?
- ☐ Wie sicher sind die verwendeten Sicherheitsmechanismen? Gibt es mögliche Angriffe dagegen?

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 endianness,
    uint32_t latencyScrambling,
    uint32_t latencyEncrypt,
    uint32_t latencyMemoryAccess,
    uint32_t seed,
    uint32_t numRequests,
    struct Request* requests,
);
```

In dieser Funktion wird das `SECURE_MEMORY_UNIT` 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 erkannten Pairty-Fehler. 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 r;
    uint8_t w;
    uint32_t fault;
    uint8_t faultBit;
};
```

Diese Requests sollten durch die Secure-Memory Unit nacheinander abgearbeitet werden. Dafür werden die entsprechenden Werte nach der Abarbeitung der vorherigen Request am Modul angelegt. Im Fall von Lesevorgängen sollen die gelesenen Daten in `data` abgespeichert 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	Fault	Fault-Bit
R	0x100			
F			0x100	6
W	123	15	0x100	5
⋮	⋮	⋮	⋮	⋮

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 bestimmt die gewünschte Operation: `R` für einen Lesezugriff, `W` für einen Schreibzugriff und `F` für eine Fault Injection. Die zweite und dritte Spalte geben die Adresse an, auf die der Speicherzugriff stattfinden soll und den Wert, der geschrieben werden

soll. Bei Lesezugriffen muss dieser Wert immer leer sein. Adresse und Daten können als Dezimalzahlen oder als Hexadezimalzahlen angegeben werden. Wenn eine Fault Injection durchgeführt werden soll, wird in der vierten Spalte die Adresse angegeben, auf die die Fault Injection angewendet werden soll und Spalte 5 gibt an, welches Bit betroffen ist. Auch die Fault Adresse darf als Dezimal- oder Hexadezimalzahl angegeben werden. Das Fault Bit muss immer als Dezimalzahl angegeben werden. Bei `F`-Requests müssen die Spalten 2 und 3 leer bleiben.

Es können auch Fault Injections während eines Lese- oder Schreibzugriffs durchgeführt werden. In diesem Fall werden die Spalten 4 und 5 gleich wie bei `F`-Requests gesetzt. Wenn keine Fault Injection durchgeführt werden soll, müssen die jeweiligen Spalten leer sein.

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