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

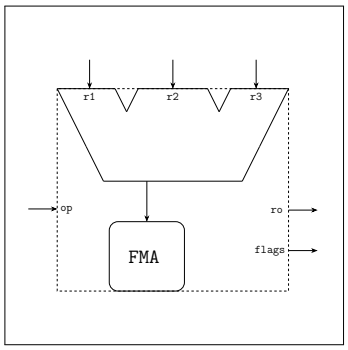
Die Abgabe muss ein `Makefile` im Wurzelverzeichnis enthalten, das über `make project` das Projekt kompilieren und die ausführbare Datei `project` erzeugen kann. Außerdem darf die Abgabe ein Shell-Script `build.sh` definieren, das den Build-Prozess startet. Dieses Build-Script wird von Artemis automatisch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



Floating-Point Unit & FMA

Ermöglicht Berechnungen mit Fließkommazahlen.

- ☐ Bietet einen neuen Datentyp für Berechnungen.
- ☐ Führt zusätzliche Operationen ein.
- ☐ Beinhaltet beschleunigte Operationen für bestimmte Berechnungen.

Durch die steigende Beliebtheit von KI und Machine Learning steigt auch die Nachfrage nach spezialisierter Hardware, die die dafür benötigten Berechnungen effizient durchführen können. Eine wichtige Funktionalität, die vor allem in diesem Bereich wichtig ist, ist **Fused Multipy-Add** (FMA): Eine Operation, die Additionen und Multiplikationen der Form

$$a \leftarrow a + (b \times c)$$

effizient in einem Schritt durchführt. Diese Operation ist besonders in der Berechnung von Neuronalen Netzen wichtig.

FMA kommt vor allem in Verbindung mit Fließkommazahlen zum Einsatz. Viele Rechnerarchitekturen, darunter auch RISC-V, bieten **Floating-Point-Extensions** an, die spezielle Instruktionen für Fließkommazahlen bereitstellen. In dieser Aufgabe definieren wir eine solche Extension im Modul `FLOATING_POINT_UNIT`, die Grundrechenarten für Fließkommazahlen bereitstellt und FMA als beschleunigte Operation implementiert.

Hinweis: RISC-V verwendet für seine Floating-Point Extensions eigene Floating Point Register, die unabhängig von den Integer-Registern sind. In dieser Aufgabe machen wir keine Unterscheidung für Registertypen und verwenden nur generelle 32-Bit Register.

SPEZIFIKATION: `FLOATING_POINT_UNIT`

Inputs

- ☐ `clk`: `bool (clock input)`
- ☐ `r1`: `uint32_t`
- ☐ `r2`: `uint32_t`
- ☐ `r3`: `uint32_t`
- ☐ `op`: `sc_bv<4>`

Outputs

- ☐ `ro`: `uint32_t`
- ☐ `zero`: `bool`
- ☐ `sign`: `bool`
- ☐ `overflow`: `bool`
- ☐ `underflow`: `bool`
- ☐ `inexact`: `bool`
- ☐ `nan`: `bool`

Implementation
Das `FLOATING_POINT_UNIT` Modul soll folgende Operationen für Fließkommazahlen implementieren:

Name	`op` Value	Result
`FADD`	8	`r1 + r2`
`FSUB`	9	`r1 - r2`
`FMUL`	10	`r1 * r2`
`FMIN`	13	min(`r1`, `r2`)
`FMAX`	14	max(`r1`, `r2`)
`FMA`	15	`r1 + (r2 * r3)`

Zu jedem Clockzyklus wird der Befehl aus `op` gelesen, der die Operation angibt, die durchgeführt werden soll. Die benötigten Operanden werden aus den jeweiligen Inputs ausgelesen und die Operation wird durchgeführt. Das Ergebnis wird in `ro` gespeichert. Überlegt genau: Wie müssen die Rechenoperationen mit dem Floating-Point Standard umgehen?

Die FMA Operation berechnet die zwei Operationen (Multiplikation und Addition) in einem Clockzyklus. Zwischen der Multiplikation und der Addition darf aber nicht gerundet werden. Das Ergebnis der Multiplikation muss also mehr als 4 Bytes zwischenspeichern. Erst nach zusätzlichem Ausführen der Addition wird das Ergebnis auf 4 Bytes gerundet. Außerdem müssen, je nach Ergebnis der Operation, die folgenden Flags gesetzt werden:

- ☐ `zero`: Das Ergebnis der Operation ist `0`.
- ☐ `sign`: Das Ergebnis ist negativ.
- ☐ `overflow`: Das Ergebnis der Operation ist größer als der größte darstellbare Wert im Positiven (`+INF`) oder im Negativen (`-INF`).
- ☐ `underflow`: Das Ergebnis der Operation ist ist näher an 0 als der Floating-Point Standard darstellen kann.
- ☐ `inexact`: Das Ergebnis der Operation konnte nicht exakt dargestellt werden und wurde deshalb gerundet.
- ☐ `nan`: Das Ergebnis der Operation ist ungültig und nicht `+INF` oder `-INF`.

Floating-Point Standard

Der Floating-Point Standard für dieses Projekt basiert auf das *IEEE 754 Single-Precision Format*, das die Darstellung von Fließkommazahlen als 32-Bit Werte definiert. Zahlen in diesem Format bestehen aus drei Teilen: Einem Vorzeichenbit (`sign`), einem Exponenten (`exponent`) und einer Mantisse (`mantissa`).

Das Vorzeichen wird mit mur einem Bit dargestellt: `0` für positive Zahlen und `1` für negative Zahlen.

Der Rest des Werts besteht aus der Fließkommadarstellung der jeweiligen Zahl im Binärformat. So wird die Zahl 3.25 im Binärformat beispielsweise als $0b11.01$ dargestellt. Alternativ kann diese Zahl auch als $0b11.01 \times 2^0 = 0b1.101 \times 2^1$ dargestellt werden. Für jede Zahl

kann der Exponent beliebig erhöht oder gesenkt werden, bis links vom Komma nur noch eine 1 übrig bleibt. Somit ergibt sich für dieses Beispiel ein Exponent von 1. Der Teil der Fließkommadarstellung nach dem Komma bestimmt die Mantisse, in diesem Fall 101.

Zur Darstellung von Fließkommazahlen wird die Mantisse mit 0 aufgefüllt, bis eine Länge von `--size-mantissa`* erreicht ist. Der Exponent wird mit $2^{(\texttt{--size-exponent}-1)}-1$ addiert, um auch negative Exponenten effizient darstellen zu können. Somit ergibt sich für die Zahl 3.25 mit `--size-exponent`=8 und `--size-mantissa`=23 eine Float-Darstellung von `0b0 10000000 10100000000000000000000`.

Es gelten auch noch einige Sonderregelungen:

- 0 wird durch Mantisse und Exponenten von 0 dargestellt.
- `+INF` und `-INF` werden mit einem Exponenten aus nur Einsen und einer Mantisse aus nur Nullen dargestellt.
- `Not a Number` wird mit einem Exponenten aus nur Einsen und einer beliebigen Mantisse $\neq 0$ dargestellt.
- `INF+INF` ergibt `INF`, `INF-INF` ergibt `NaN`.
- `INF*0` ergibt `NaN`.
- `0*NaN` ergibt `0`, alle anderen Operationen mit `NaN` ergeben `NaN`.
- Ergebnisse von Operationen, die über `INF` hinaus gehen, sollen auf `INF` gerundet werden.
- Ergebnisse von Operationen, die zu klein für die Float-Darstellung sind, sollen auf 0 gesetzt werden.

Außerdem muss beachtet werden, dass nicht jede Operation exact durchgeführt werden kann. In diesem Fall muss das Ergebnis entsprechend `--round-mode`* gerundet werden:

- `0 (Round to nearest, ties to even)`: Das Ergebnis wird auf die nächstgelegene Zahl gerundet. Wenn die letzte Ziffer 5 ist, wird in die Richtung der nächsten geraden Ziffer gerundet.
- `1 (Round to nearest, ties away from zero)`: Das Ergebnis wird auf die nächstgelegene Zahl gerundet. Wenn die letzte Ziffer 5 ist, wird in die Richtung gerundet, die weiter von 0 entfernt ist.
- `2 (Round toward zero)`: Das Ergebnis wird in die Richtung von 0 gerundet.
- `3 (Round toward +INF)`: Das Ergebnis wird in die Richtung von `+INF` gerundet.
- `4 (Round toward -INF)`: Das Ergebnis wird in die Richtung von `-INF` gerundet.

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

- ☐ `uint32_t getPositiveInf()`:

  Gibt die Floating-Point Darstellung von ***`+INF`*** *zurück.*

- ☐ `uint32_t getNegativeInf()`:

  Gibt die Floating-Point Darstellung von ***`-INF`*** *zurück.*

- ☐ `double getMax()`:

  Gibt den höchstmöglichen Wert mit den momentanen Optionen als ***`double`*** *zurück.*

- ☐ `double getMin()`:

  Gibt den kleinstmöglichen positiven Wert mit den momentanen Optionen als ***`double`*** *zurück.*

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

Erlaubte *Magie*: Rechenoperationen, Kontrolle von Flags, Runden von Fließkommazahlen, Erstellen von Fließkommazahlen

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.*

- ☐ `--size-exponent`: `uint8_t` — *Die Anzahl der Bits, die für den Exponenten verwendet werden sollen.*
- ☐ `--size-mantissa`: `uint8_t` — *Die Anzahl der Bits, die für die Mantisse verwendet werden sollen.*
- ☐ `--round-mode`: `uint8_t` — *Der gewünschte Rundungsmodus.*

Weitere Hinweise

- ☐ `--size-exponent` und `--size-mantissa` sind nur gültig, wenn ihre Summe nicht zu groß oder zu klein für 32 Bits ist.
- ☐ Überlegt genau, wann welche Flags gesetzt werden sollten.
- ☐ Die `FLOATING_POINT_UNIT` kann auch in das `TINY_RISC` Modul eingebaut werden. Dafür muss das Instruction Set in der `CU` erweitert werden. Sie kann dann mit der `ALU` verbunden werden.
- ☐ Das gesamte Floating Point Modul kann und soll aus Untermodulen (z.B. für die FMA Erweiterung) bestehen.

Fragen für die Literaturrecherche

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

- ☐ Warum ist FMA besonders für die Berechnung von Neuronalen Netzen wichtig?
- ☐ Welche andere Verwendungen hat FMA?
- ☐ Was sind die Minimal- und Maximalwerte für Floats mit verschiedenen Werten für `--size-exponent` und `--size-mantissa`?

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 sizeExponent,
    uint8_t sizeMantissa,
    uint8_t roundMode,
    uint32_t numRequests,
    struct Request* requests,
);
```

In dieser Funktion wird das `FLOATING_POINT_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 signs;
    uint32_t overflows;
    uint32_t underflows;
    uint32_t inextacts;
    uint32_t nans;
};
```

Dieses Struct beinhaltet Statistiken der Simulation, wie die Anzahl der zur Abarbeitung benötigen Zyklen, die Anzahl der erhaltenen `sign`, `overflow`, `underflow`... Flags. 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 r1; uint32_t r2; uint32_t r3;
    uint32_t ro;
    uint8_t op;
};
```

Diese Requests sollten durch die Floating Point Unit nacheinander abgearbeitet werden. Dafür werden die Inputs `r1`, `r2`, `r3` und `op` am Modul angelegt und der erhaltene Ausgabewert wird in `ro` abgespeichert.

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:

op	r1	r2	r3
9	0x00000011	0x10001011	
13	0x00000001	32949138	11112
15	0.001	2.395	0xff
⋮	⋮	⋮	⋮

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, die restlichen Spalten sind die Operanden. Die Operanden können als Fließkommazahlen oder Hexadezimalzahlen angegeben werden. Außerdem können sie als 32-Bit Ganzzahlen angegeben werden und ihre Binärdarstellung wird dann als Fließkommazahl interpretiert. Fließkomma-Operanden müssen in das durch die CLI-Optionen definierte Float-Format übersetzt werden, bevor

sie in die Felder von `Request` Structs gesetzt werden können. Die Operation muss nur als Dezimalzahl eingelesen werden. Wenn ein Operand nicht benötigt wird, kann das dazugehörige Feld auch leer gelassen werden.

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