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.

Die Aufgabenstellung definiert eine Erweiterung der SystemC Module und des Assemblierers, die im Rahmen der Hausaufgaben erstellt wurden. Diese Erweiterung benötigt möglicherweise Änderungen an der existierenden Struktur und Kommunikation zwischen den Modulen. Besprecht deshalb innerhalb eurer Gruppe, welches Abstraktionsniveau für die Implementierung sinnvoll ist und diskutiert den Entwurf der Erweiterungen 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 werden.

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

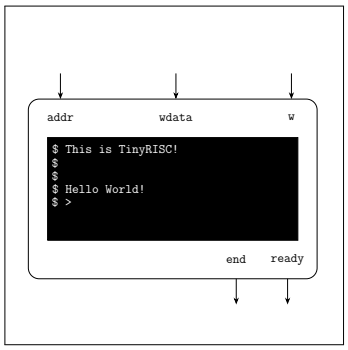
Die Abgabe muss ein `Makefile` im Wurzelverzeichnis enthalten, das über `make project` das Projekt und über `make assembler` den Assemblierer kompilieren und die ausführbare Datei `project/assembler` erzeugen kann. Außerdem darf die Abgabe ein Shell-Script `build.sh` definieren, das den Build-Prozess 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 in diesem Dokument genannten Vorgaben 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.
- `libs/` — Ein Unterordner, der Assembly-Libraries (und spezifisch `stdio.asm`) beinhaltet.
- `src/` — Ein Unterordner, der alle Quelldateien enthält.
- `src/assembler` — Ein Unterordner mit Quelldateien für den Assemblierer.
- `src/risc` — Ein Unterordner mit Quelldateien für die TinyRISC CPU.
- `include/` — Ein Unterordner, der alle Headerdateien enthält.
- `include/assembler` — Ein Unterordner mit Headerdateien für den Assemblierer.
- `include/risc` — Ein Unterordner mit Headerdateien für die TinyRISC CPU.
- `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 **genau zwei** ausführbare Dateien mit den Namen `project` und `assembler` im aktuellen Ordner erstellen. Abweichungen von dieser Vorgabe können zu Abzügen führen.



STDIO

Unterstützung für Peripheriegeräte in TinyRISC.

- ☐ Kommuniziert mit Peripheriegeräten durch Memory Mapping.
- ☐ Calling Convention ermöglicht Funktionsaufrufe.
- ☐ Standardbibliothek für TinyRISC Assembly.

Ohne Peripheriegeräte sind selbst fortgeschrittene Systeme wie unsere TinyRISC CPU nur sehr begrenzt nutzbar. In diesem Projekt soll daher eine Möglichkeit entwickelt werden, Text über eine simulierte Konsole auszugeben.

Die Speicheradresse `0xFFFFFFFF` wird über **Memory Mapping** als Terminal-Output festgelegt. Werte, die auf diese Adresse geschrieben werden, sollen direkt an das `TERMINAL` Modul weitergegeben werden.

Damit neuer Assembler-Code dieses Feature effizient nutzen kann, soll eine ASM Library entwickelt werden, die eine `print` Funktion bereitstellt. Der TinyRISC Assemblierer muss dann aktualisiert werden, um zusätzliche Libraries an gegebene Assemblerprogramme **linken** zu können.

Außerdem setzt die effiziente Verwendung einer solchen `print` Funktion auch die Unterstützung einer **Calling Convention** voraus. Damit können Funktionen mit Parametern einheitlich aufgerufen werden. Die entsprechend benötigten neuen Anweisungen müssen auch vom TinyRISC Assemblierer und der CPU selbst unterstützt werden.

SPEZIFIKATION: `TERMINAL`

Inputs	*Outputs*
☐ `clk`: `bool (clock input)`	☐ `ready`: `bool`
☐ `addr`: `uint32_t`	☐ `end`: `bool`
☐ `wdata`: `uint32_t`	
☐ `w`: `bool`	

Implementation
Zur steigenden Flanke von `clk` werden die Inputs `addr` und `w` überprüft. Wenn auf `0xFFFFFFFF` geschrieben werden soll, wird der Wert in `wdata` in das Terminal geschrieben. Dafür muss das Byte, des 4-Byte Inputs `wdata`, das in Adresse `0xFFFFFFFF` landen würde (wir nehmen an, dass Eingaben dem Little-Endian Encoding folgen), zu einem `char` konvertiert werden. Die restlichen Bytes werden ignoriert. Wenn `--terminal-file*` nicht gesetzt ist, muss der entsprechende Wert mittels `std::cout` auf die Konsole geschrieben werden. Ansonsten wird der zu schreibende Wert nur vom Terminal zwischengespeichert.

Erhält das Terminal einen Input von `0`, wird nichts ausgegeben. Stattdessen, wird der Wert von `end` bis zum nächsten Clock-Zyklus auf `1` gesetzt. Außerdem muss in diesem Fall sichergestellt werden, dass `stdout` geflusht wird, wenn es zur Darstellung der Terminal-Outputs verwendet wird.

Das Asugeben eines Outputs im Terminal benötigt `--latency*` Clock-Zyklen. Zu Beginn jeder Output Operation wird `ready` auf `0` gesetzt. Nach dem Abschluss eines Schreibvorgangs oder wenn `end` auf `1` gesetzt wird, wird `ready` zu `1`.

Die Ports `ready`, `addr`, `wdata` und `w` können mit den dazugehörigen Ports der CU verbunden werden. Da diese Ports aber auch von anderen Komponenten der TinyRISC CPU verwendet werden, sind möglicherweise neue Multiplexer notwendig. Die `ready` Signale des Terminals und des Hauptspeichers können in einem 2:1 Multiplexer verbunden werden. Die CU benötigt dann einen weiteren Output Port, der den `select` Input des Multiplexers festlegt und somit entscheidet, ob auf Terminal oder Hauptspeicher gewartet wird.

Assembly-Format Update
Der TinyAssembler aus der dazugehörigen Hausaufgabe soll um die folgende Anweisung erweitert werden:

```
jal rd, offset
```

Der aktuelle PC-Wert + 4 wird auf das Register `rd` geschrieben. Dann wird der `PC` um `offset` erhöht.
Um maximale kompatibilität mit der existierenden TinyRISC CPU zu gewährleisten, verwenden `jal` Anweisungen folgendes Format:

Bit:	31 - 20	19 - 12	11 - 7	6 - 0
Field:	offset	00000000	rd	opcode

Labels: Der TinyAssembler soll nun auch Labels im Assemblycode unterstützen. Labels sind Identifier, die aus Kleinbuchstaben, Zahlen und '_' ("Unterstrich") Symbolen bestehen und mit einem ':' beendet werden. Zeilen, die ein Label definieren, dürfen anonsten *nur* Whitespace und Kommentare beinhalten.
Beispiel:

```
jal x31, hello_world

hello_world: # This function doubles the value in x9.
    add x9, x9, x9
    j x31 # return to address before function call.
```

Alle Befehle, die einen `offset` als Immediate erwarten, dürfen nun stattdessen ein Label als Parameter annehmen. Der Assemblierer berechnet dann automatisch den benötigten Offset, um bei der nächsten Anweisung nach dem Label fortzufahren, bevor der Bytecode für die jeweilige Anweisung erstellt wird. *Hinweis:* Fälle, in denen der Abstand zwischen Verwendung und Definition eines Labels so groß sind, dass sie den Maximalwert von `offset` Immediates überschreiten, können ignoriert werden. Fälle, in denen ein Label mehrfach definiert wurde, dürfen beliebig gehandhabt werden.

TinyRISC Update
Um die neue Anweisung zu unterstützen, müssen auch manche Ports der existierenden TinyRISC CPU (spezifisch zum Schreiben von `PC + 4` auf `rd`) neu verbunden werden. Außerdem muss die Logik der CU überarbeitet werden. Die genaue Umsetzung davon bleibt euch überlassen.

`stdio` Library

Erstellt eine neue Datei namens `stdio.S`. Diese Datei muss im Abgaberepository `libs/stdio.S` liegen.

`stdio.S` definiert die Funktion `print`. Diese Funktion akzeptiert einen Parameter, `p`, der die Adresse des Starts des auszugebendne Strings im Hauptspeicher angibt. Die `print` Funktion iteriert dann, beginnend von `p`, Byte für Byte durch den Speicher. Jedes Byte wird dabei einzeln an das Terminal geschickt, indem es auf die gemappte Adresse geschrieben wird.

Ein String im Hauptspeicher muss immer mit dem Null-Terminator (`0x0`) als letztes Byte enden. Sobald `print` auf diesen Wert trifft, wird er noch als letztes an das Terminal weitergeleitet. Danach wird die Ausführung der Funktion beendet.

Calling Convention: Die `print` Funktion soll von außen mittels der Anweisung `jal x31, print` aufgerufen werden können. Nach erfolgreicher Ausführung soll der PC-Wert durch `j x31` auf die Rücksprungadresse zurückgesetzt werden. Parameter für Funktionen sollen der Reihe nach in die Register geschrieben werden, beginnend mit `x1` für den ersten (und im Fall von `print` einzigen) Parameter.

TinyAssembler: Linking

Zuletzt soll der Assemblierer `TinyAssembler` noch um eine Funktionalität erweitert werden: Eine vereinfachte Form von Linking. Er akzeptiert nun weitere optionale CLI Argumente in der Form:
`./assembler <input> [library1] [library2] ....`
Damit können Pfade zu zusätzlichen `.S` Libraries angegeben werden. Der Inhalt dieser Dateien wird dann beim Assembling-Prozess dem zu assemblierenden Code hinzugefügt, damit Funktionen, die darin implementiert wurden, vom Hauptprogramm aus aufgerufen werden können.

Die genaue Implementierung dieses Linking-Vorgangs bleibt euch überlassen. Beachtet aber, dass es sich hier nicht um einen herkömmlichen Linker handelt: Es werden keine Object-Files für die Library generiert, sondern Libraries werden im Parsing-Prozess des Assemblierers direkt als Code dem restlichen Input hinzugefügt. *Achtung:* Das `offset`

Argument der TinyRISC Anweisungen unterstützt nur positive Werte. Achtet also darauf, dass Funktionen von Libraries an eine spätere Position im Bytecode gesetzt werden, als der Code des Hauptprogramms. (Es darf davon ausgegangen werden, dass sich verschiedene Libraries nicht gegenseitig aufrufen.)

Ein Assemblerprogramm, das die `print` Funktion der `stdio.S` Library aufruft, muss dem Assemblierer also wie folgt übergeben werden:
`./assembler my-code.S stdio.S`

Methoden
Das `TERMINAL` Modul stellt außerdem die folgende Methode zur Verfügung:

- ☐ `void output(char value)`:

  Gibt einen einzelnen ***`char`*** *Wert aus.*

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

Weitere Aufgaben für die Präsentation
Für die Bewertung der Implementierung des Projekts muss die `stdio.S` Library nur die oben angegeben `print` Funktion unterstützen. Allerdings sollte eure Abgabe *mindestens eine weitere passende Funktion* beinhalten, die von `stdio.S` bereitgestellt wird. Stellt diese Funktion in der Präsentation kurz vor.

Erlaubte *Magie*: `jal` Anweisung decoden, Prüfen des Memory Mapping Adresse, Ausgaben im Terminal, Speichern von Werten und Flags.

Die Abgabe beinhaltet Code für zwei ausführbare Dateien: `assembler` und `project`. `assembler` ist die überarbeitete Version des TinyAssemblers aus den Hausaufgaben mit der neuen Funktionalität für das Projekt. `project` ist die Hauptdatei des Projekts. Es startet die Simulation der vollständigen TinyRISC CPU, die um das `TERMINAL` Modul erweitert wurde. Außerdem unterstützt die `project`-Executable folgende CLI Optionen:

Optionen*

Die folgende Liste zählt alle zusätzlichen Optionen auf, die vom Rahmenprogramm erkannt und angewendet werden müssen.

- ☐ `--terminal-file`: `const char*` — *Gibt den Pfad an, der für den Terminal Output verwendet werden soll. Wenn diese Option nicht angegeben ist, schreibt das Terminal auf* ***`stdout`****.*
- ☐ `--latency`: `uint32_t` — *Gibt die Anzahl an Clock-Zyklen an, die für die Ausgabe eines Werts am Terminal benötigt werden.*

Weitere Hinweise

- ☐ Das Einhalten der vorgegebenen Ordnerstruktur bei der Abgabe des Projekts ist sehr wichtig!
- ☐ Dieses Projekt interagiert direkt mit den in den Hausaufgaben behandelten Modulen.
  - ☐ Alle nötigen Dateien müssen abgegeben werden, das Projekt und der Assemblierer müssen sofort kompilierbar sein.
  - ☐ Die Musterlösungen der Hausaufgaben oder die Abgaben von Gruppenmitgliedern dürfen als Grundlage für das Projekt verwendet werden.

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

- ☐ Die hier verwendete Calling Convention ist sehr minimalistisch. Was könnte beispielsweise schiefgehen, wenn verschachtelte Funktionen aufgerufen werden?
- ☐ Wie wird das vorherige Problem häufig gelöst?
- ☐ Wie unterscheidet sich der vereinfachte Linker in diesem Projekt von herkömmlichen Linkern?

Rahmenprogramm

Für die CPU Simulation soll ein Rahmenprogramm in C implementiert werden. 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 my_program.out
```

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* instructions,
    char terminalToStdout,
    uint32_t latency
);
```

In dieser Funktion wird das `TinyRISC` Modul initialisiert und die Simulation gestartet. Außerdem müssen die entsprechenden Werte der Anweisungen in den Hauptspeicher geladen werden, damit die Ausführung korrekt durchgeführt wird. `terminalToStdout` gibt

an, ob das `TERMINAL` Modul auf `stdout` schreiben soll. Falls `--terminal-file` angegeben wurde, wird `terminalToStdout` auf `0` gesetzt. Die Ergebnisse der Simulation sollen in einem `Result` Struct zurückgegeben werden.

```
struct Result {
    uint32_t cycles;
    char* output;
    uint32_t outs;
};
```

Dieses Struct beinhaltet Statistiken der Simulation, wie die Anzahl der zur Abarbeitung benötigen Zyklen. Im `outs` Feld wird angegeben, wie viele Outputs im Terminal durchgeführt wurden. Dafür kann gezählt werden, wie oft die `end` Flag auf `1` gesetzt wurde. Falls `--terminal-file` angegeben wurde, enthält `output` alle gesammelten Outputs des `TERMINAL` Moduls. Diese müssen vom Rahmenprogramm aus dann in die angegebene Datei geschrieben werden. Die restlichen wichtigen Informationen des `Result` Structs sollten nach der Ausführung in der Kommandozeile anschaulich ausgegeben werden.

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

Eingabedatei

Die Eingabedatei beinhaltet ein bereits assembliertes, ausführbares Programm für die TinyRISC CPU. Die Datei listet auszuführende Anweisungen im Binärformat (ohne führendes "`0b`") auf, die durch Zeilenumbrüche getrennt sind. Sie soll im Rahmenprogramm eingelesen werden und der `run_simulation` Funktion als Buffer vom Typ `uint32_t*` übergeben werden. Der PC der TinyRISC CPU beginnt bei `0x00001000`, also müssen die eingelesenen Anweisungen an die passende Adresse geschrieben werden, damit das Programm ausgeführt werden kann. (Der `TINY_RISC` Konstruktor akzeptiert das Argument `initialMemory`, worüber Werte, die zu Beginn der Simulation in den Hauptspeicher geschrieben werden sollen, übergeben werden können).