Lernziele:
- Erste Schritte in der Assembler-Programmierung
- Umgang mit dem ULM-Generator

Vorbemerkung
In dieser Übung verwenden wir die **ulm-ice40**-Architektur und die Werkzeuge, die dafür mit dem ULM-Generator aus **ulm-ice40.isa** erstellt werden. Installationsanleitungen finden sich auf der Vorlesungsseite zu dieser Session.

Schritt 1: Umgang mit den generierten Tools
Wir beginnen mit einem einfachen Assembler-Programm **ex1.s** (beachtet, dass jede Zeile mit mindestens einem Whitespace eingerückt sein sollte):

```
    getc    %1
    putc    %1
    halt    0
```

Dieses Programm liest ein Zeichen aus dem Tastaturpuffer in Register **%1**, gibt dann das in Register **%1** gespeicherte Zeichen aus und beendet das Programm mit dem Exit-Code **0**.

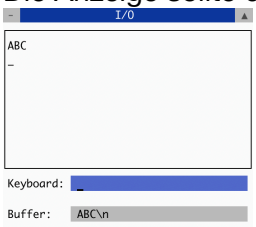
Aufgaben:
1. Übersetzt dieses Programm mit **ulmas -o ex1 ex1.s** in eine auf der ULM ausführbare Datei **ex1**. Bestätigt, dass **ex1** aus folgenden Zeilen besteht:

```
#TEXT 4
0x0000000000000000: 32 10 00 00 #     getc %1
0x0000000000000004: 30 10 00 00 #     putc %1
0x0000000000000008: 01 00 00 00 #     halt 0
#DATA 1
#BSS 1 0
#SYMTAB
```

2. Ladet das Programm mit **udb-tui ex1** im ULM-Debugger und macht euch mit den Anzeigen für den Speicherinhalt und der Register vertraut. Im Speicher sollten die ersten 12 Bytes mit dem Programm initialisiert sein:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x0000000000000000	32	10	00	00	30	10	00	00	01	00	00	00	00	00	00	00

   Bevor das Programm gestartet wird, könnt ihr den Tastaturpuffer bereits füllen. Gebt dazu beim IO-Device im Eingabefeld zu **Keyboard** den Text "ABC" gefolgt von einem Return ein. Die Anzeige sollte dann wie folgt aussehen:



   Wenn ihr nun mit **Control-S** genau eine Instruktion ausführt, solltet ihr folgendes beobachten:
   1. Der Instruction-Pointer **%IP** wird um 4 erhöht.
   2. Im Instruction-Register **%IR** steht der Maschinenbefehl für **getc %1**.
   3. Aus dem Tastaturpuffer wurde das Zeichen **'A'** entfernt.
   4. In Register **%1** steht der ASCII-Code von **'A' (0x41)**.
   5. Führt schrittweise weitere Befehle aus, bis das Programm terminiert. Beobachtet dabei, was durch jede Instruktion verändert wurde.
3. Beendet den Debugger und startet ihn neu. Führt diesmal das Programm aus, ohne vorher den Tastaturpuffer zu füllen.
4. Führt das Programm im Terminal mit **ulm ex1** aus.

Schritt 2: Sprunganweisungen und Labels
Erweitert das Programm "ex1.s" um eine unbedingte Sprunganweisung wie folgt:

```
    getc    %1
    putc    %1
    jmp     -8
    halt    0
```

Nach der Ausgabe eines Zeichens springt die Sprunganweisung zu der Instruktion, die 8 Bytes vorher im Speicher liegt, das ist die Instruktion, die ein Zeichen einliest.

Aufgaben:

1. Übersetzt das Programm wieder in **ex1** und führt es zunächst mit **ulm ex1** (wer möchte, kann das Programm natürlich auch im Debugger ausführen) aus. Gibt man nun zeilenweise Text ein, dann wird dieser anschließend vom Programm ausgegeben:

   ```
   MCL:hpc0 lehn$ ulm ex1
   hallo
   hallo
   welt
   welt
   ^C
   ```

   Da die Halt-Instruktion nicht erreicht wird (ggf. den Kontrollfluss mit einem Flow-Chart visualisieren), ist es notwendig, die Ausführung mit Control-C zu beenden.
2. Betrachtet den Inhalt der ausführbaren Datei **ex1**:

   ```
   #TEXT 4
   0x0000000000000000: 32 10 00 00 #     getc %1
   0x0000000000000004: 30 10 00 00 #     putc %1
   0x0000000000000008: 05 FF FF FE #     jmp -8
   0x000000000000000C: 01 00 00 00 #     halt 0
   #DATA 1
   #BSS 1 0
   #SYMTAB
   ```

   Die Jump-Instruktion **jmp -8** wurde mit **05 FF FF FE** codiert. Der Wert **-8** wurde dabei mit dem 24-stelligen Bitmuster **FF FF FE** codiert, was dem Signed-Wert **-2** entspricht. Bestätigt, dass zum Beispiel **jmp 8** mit **05 00 00 02** codiert werden würde, der Wert **8** in der Assembler-Instruktion also in der Instruktion mit dem Wert **2** codiert wird. Erkläre, wieso es allgemein sinnvoll ist, dass dieses "Sprungoffset" vom Assembler immer durch **4** geteilt wird und dann dieser Wert in die Instruktion codiert wird.
3. Ändert das Programm in:

   ```
   G   getc    %1
       putc    %1
       jmp     G
       halt    0
   ```

   Hier wird das Label **G** verwendet. Der Wert von **G** ist die Adresse der Instruktion **getc %1**. Bestätigt, dass der erzeugte Maschinencode identisch ist mit der vorigen Version. In der ausführbaren Datei enthält die Symboltabelle jetzt aber einen Eintrag für das Symbol **G**:

   ```
   t G                     0x0000000000000000
   ```

4. Das Programm soll nun terminieren, wenn ein **EOF** (End of File) eingelesen wird. Ein **EOF** kann im Terminal mit **Control-D** erzeugt werden. Bei der ulm-ice40-Architektur wird **EOF** mit 0xFF codiert. Testet folgende Programmänderung:

   ```
   G   getc    %1
       subq    0xFF,   %1,     %0
       jz      H
       putc    %1
       jmp     G
   H   halt    0
   ```

   Durch die "subq"-Instruktion in Zeile 2 wird das ZF-Flag gesetzt, wenn in Register %1 ein EOF eingelesen wurde. In diesem Fall wird mit der "jz"-Instruktion (für "Jump Zero") zur Halt-Instruktion gesprungen.

5. Das Programm soll nun eine nicht-negative Zahl in Dezimaldarstellung einlesen und dann mit dem Wert dieser Zahl als Exit-Code terminieren. Ändert dazu das Programm in:

```
 1      loadz   0,      %2
 2
 3 G    getc    %1
 4      subq    0xFF,   %1,     %0
 5      jz      H
 6      subq    '9',    %1,     %0
 7      ja      H
 8      subq    '0',    %1,     %0
 9      jb      H
10      mulw    10,     %2,     %2
11      subq    '0',    %1,     %1
12      addq    %1,     %2,     %2
13      jmp     G
14
15 H    halt    %2
```

   Bevor wir uns mit den Details der Änderung beschäftigen, sollte das Programm getestet werden:
   1. Übersetzt das Programm und gebt bei der Ausführung den Text "123" gefolgt von einem Return ein. Überprüft dann mit **echo $?**, dass der Exit-Code tatsächlich den Wert **123** hat.
   2. Führt das Programm ein weiteres Mal aus und gebt einen Wert ein, der größer als 255 ist. Der Exit-Code scheint dann scheinbar nicht zu stimmen. Gibt man zum Beispiel "1234" ein, dann ist der Exit-Code 210. Der Grund dafür ist, dass unter Unix für den Exit-Code nur 8 Bits verwendet werden. Ein Wert wie 1234 wird also auf 8 Bits abgeschnitten, was dann dem Wert $1234 \bmod 2^8 = 1234 \bmod 256 = 210$ entspricht.
6. Details im vorigen Assembler-Programm:
   1. Die Sprunganweisung "ja" (für "Jump Above") in Zeile 7 führt den Sprung aus, wenn bei der Subtraktion in Zeile 6 der Wert '9' größer als der Wert in Register %1 war.
   2. Die Sprunganweisung "jb" (für "Jump Below") in Zeile 9 führt den Sprung aus, wenn bei der Subtraktion in Zeile 8 der Wert '0' kleiner als der Wert in Register %1 war.
   3. Als ABC-Programm kann das vorige Assembler-Programm wahlweise mit folgenden Programmen ausgedrückt werden (der Compiler erzeugt aus beiden Varianten den gleichen Maschinencode). Die Variante links drückt dabei den obigen Assembler-Code besser aus, die Variante rechts dafür besser die Semantik:

```
@ <stdio.hdr>

fn main(): unsigned
{
    local val: int = 0;
    local ch: int;

label G:
    ch = getchar();
    if (ch == EOF) { goto H; }
    if (ch < '0') { goto H; }
    if (ch > '9') { goto H; }
    val *= 10;
    val += ch - '0';
    goto G;

label H:
    return val;
}
```

```
@ <stdio.hdr>

fn main(): int
{
    local val: int = 0;
    local ch: int;

    while ((ch = getchar()) != EOF) {
        if (ch < '0' || ch > '9') {
            break;
        }
        val = val * 10 + ch - '0';
    }
    return val;
}
```

Aufgabe: Optimierung des Codes
Optimiert den Code des Assembler-Programms, indem ihr folgende Änderungen durchführt: In Zeile 8 soll das Ergebnis in Register %1 statt Register %0 geschrieben werden und Zeile 11 soll gestrichen werden. Testet das Programm und erklärt eurem Nachbarn, wie diese Optimierung funktioniert.

Schritt 3: Einfache Funktionen (Leaf-Funktionen)

In folgendem Programm **ex2.s** wurden zwei einfache Funktionen **foo** (in den Zeilen 15 bis 20) und **bar** (in den Zeilen 22 bis 27) realisiert, die jeweils den Text "foo" und "bar" ausgeben:

```
 1     loadz   foo,    %1
 2     call    %1,     %2
 3
 4     loadz   bar,    %1
 5     call    %1,     %2
 6
 7     loadz   foo,    %1
 8     call    %1,     %2
 9
10     loadz   bar,    %1
11     call    %1,     %2
12
13     halt    0
14
15 foo:
16     putc    'f'
17     putc    'o'
18     putc    'o'
19     putc    '\n'
20     ret     %2
21
22 bar:
23     putc    'b'
24     putc    'a'
25     putc    'r'
26     putc    '\n'
27     ret     %2
```

Aufgaben:

1. Übersetzt das Programm in "ex2" und führt es anschließend mit ulm ex2 aus. Dies sollte folgende Ausgabe erzeugen:

   ```
   MCL:hpc0 lehn$ ulmas -o ex2 ex2.s
   MCL:hpc0 lehn$ ulm ex2
   foo
   bar
   foo
   bar
   ```

2. Führt das Programm im Debugger aus. Dabei sollt ihr folgendes beobachten:
   1. Mit der "loadz"-Instruktion in Zeile 1 wird die Adresse der ersten Instruktion der Funktion "foo" in das Register %1 geladen.
   2. Mit der "call"-Instruktion in Zeile 2 wird zu der in Register %1 hinterlegten Adresse gesprungen. Gleichzeitig wird die Rücksprungadresse in Register %2 hinterlegt. Die Rücksprungadresse ist dabei die Adresse der Instruktion in Zeile 4.
   3. Wird in der Funktion "foo" die Instruktion "ret %2" ausgeführt, erfolgt ein Sprung zur in Register %2 hinterlegten Rücksprungadresse.
   4. Führt das Programm weiter instruktionsweise aus, um zu beobachten, dass ein Funktionsaufruf stets aus dem Laden der Funktionsadresse und einer anschließenden "call"-Instruktion besteht, und dass beim Verlassen der Funktion stets an die richtige Rücksprungadresse gesprungen wird.
3. Es ist unnötig, dass zwei Register (%1 und %2) für die Funktionsaufrufe verwendet werden. Ändert die "call"-Instruktionen ab in "call %1, %1" und die "ret"-Instruktionen in "ret %1".

Bemerkung: Funktionen, die keine anderen Funktionen aufrufen, nennt man "Leaf-Funktionen".

Schritt 4: Erweiterung der Architektur

In diesem Schritt soll gezeigt werden, wie die Architektur um weitere Befehle erweitert werden kann. Fügt dazu in **ulm-ice40.isa** am Ende folgende Zeilen ein:

```
0x0E    RR
:   call    %x
    ulm_absJump(ulm_regVal(x), x);

0x0F    U20_R
:   call    imm, %dest
    ulm_absJump(imm, dest);
```

Erzeugt und installiert die erweiterte Architektur dann mit **ulm-generator --install ulm-ice40.isa** neu. Es stehen nun zwei neue Instruktionen zur Verfügung, die wie folgt verwendet werden können:

1. Mit **call %1** wird die in Register %1 hinterlegte Adresse angesprungen und gleichzeitig in Register %1 die Rücksprungadresse hinterlegt.
2. Mit **call foo, %1** wird die Adresse foo angesprungen und gleichzeitig in Register %1 die Rücksprungadresse hinterlegt. Bei dieser Variante ist es allerdings notwendig, dass die Adresse foo mit 20 Bits als Unsigned-Integer dargestellt werden kann (also im Bereich von 0 bis 1048575 liegt).

Aufgabe:

Ändert das Programm so, dass nun alle drei Varianten der "call"-Instruktionen verwendet werden. Bestätigt, dass bei den erzeugten Maschinenbefehlen unterschiedliche Op-Codes verwendet werden, es also technisch betrachtet um völlig unterschiedliche Instruktionen handelt.

Falls du eine funktionierende LaTeX-Installation hast, erzeuge ein Reference-Manual mit dem Kommando **ulm-generator --refman ulm-ice40.isa**.

Schritt 5: Vorbereitung auf Non-Leaf-Funktion

Das nächste Ziel ist es, dass Funktionen auch andere Funktionen aufrufen können. Bestätige, dass folgender Ansatz (bei dem die Funktion "foo" die Funktion "bar" aufruft) nicht funktioniert:

```
    call    foo,    %1
    halt    0

foo:
    putc    'f'
    putc    'o'
    putc    'o'
    putc    '\n'

    call    bar,    %1

    ret     %1

bar:
    putc    'b'
    putc    'a'
    putc    'r'
    putc    '\n'
    ret     %1
```

Erkläre deinem Nachbarn, warum dieser Ansatz nicht funktionieren kann, und warum das völlig unabhängig von der verwendeten call-Instruktion ist.

Schritt 6: Verwendung eines Stacks für Funktionsaufrufe

In folgendem Programm "ex3.s" wurde ein Konzept realisiert, das es Funktionen erlaubt, andere Funktionen aufzurufen:

```
    loadz   0,      %2

    call    foo,    %1
    halt    0

foo:
    subq    8,      %2,     %2
    movq    %1,     (%2)

    putc    'f'
    putc    'o'
    putc    'o'
    putc    '\n'
    call    bar,    %1

    movq    (%2),   %1
    addq    8,      %2,     %2
    ret     %1

bar:
    subq    8,      %2,     %2
    movq    %1,     (%2)

    putc    'b'
    putc    'a'
    putc    'r'
    putc    '\n'

    movq    (%2),   %1
    addq    8,      %2,     %2
    ret     %1
```

Aufgaben:

1. Übersetze das Programm in "ex3" und führe es mit ulm ex3 aus, um zu bestätigen, dass dieser Ansatz funktioniert.
2. Im Programm sind folgende Änderungen wesentlich:
   1. In Zeile 1 wurde das Register %2 mit 0 initialisiert.
   2. Funktionen beginnen bei diesem Ansatz immer mit den Zeilen:

      ```
      subq    8,      %2,     %2
      movq    %1,     (%2)
      ```

      Dabei ist movq %1, (%2) eine Store-Operation, die den Inhalt von Register %1 an die in Register %2 hinterlegte Adresse kopiert.
   3. Funktionen enden immer mit:

      ```
      movq    (%2),   %1
      addq    8,      %2,     %2
      ret     %1
      ```

      Dabei ist movq (%2), %1 eine Fetch-Instruktion, die von der in Register %2 hinterlegten Adresse 8 Bytes in das Register %1 kopiert.

   Benutze den Debugger, um zu erkunden, wie es damit ermöglicht wird, dass Funktionen andere Funktionen aufrufen können.
3. Schreibe ein Assembler-Programm mit einer Funktion "foo", die den Text "foo" ausgibt und sich dann abhängig vom Wert in Register %3 aufruft:
   1. Wenn das Register %3 den Wert 0 enthält, soll kein rekursiver Aufruf erfolgen. Andernfalls soll der Wert in Register %3 um 1 dekrementiert werden und ein rekursiver Aufruf durchgeführt werden.
   2. In der ersten Zeile des Programms soll das Register %3 mit dem Wert 42 initialisiert werden.