Lernziele:
- Einfache Funktionen (wie in Session 10) in Assembler realisieren
- Umgang mit String-Literalen
- Pattern für einfache Funktionsaufrufe erweitern für Parameterübergabe und Rückgabewerte
- Vorbereitung und Motivation für ein Pattern das Stack-Pointer und Frame-Pointer verwendet
- Umgang mit dem ULM-Generator

Vorbemerkung
In dieser Übung verwenden wir die Architektur, die in Session 10 verwendet wurde. Zur Erinnerung: In Session 10 wurde die ISA `ulm-ice40.isa` erweitert, um Funktionsaufrufe zu vereinfachen. In die Datei `ulm-ice40.isa` wurden dazu folgende Zeilen eingefügt:

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

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

Bisheriges Pattern für Funktionen

In Session 10 wurde zudem ein Pattern für den Umgang mit einfachen Funktionen (die Funktionen haben keine Parameter und keinen Rückgabewert) erarbeitet. Zur Verwaltung der Rücksprungadresse wird ein Stack verwendet. Als Stack-Pointer (`SP`) wird Register `%1` verwendet:

```
        .equ    SP,         1
```

und nach dem Programmstart initialisiert:

```
        loadz   0,          %SP
```

Das Pattern beschreibt zwei Konzepte:

- **Aufruf einer Funktion:** Kann mit einer Call-Instruktion umgesetzt werden, die in einem Register die Rücksprungadresse hinterlegt. Beispielsweise wird mit:

  ```
          call    foo,        %2
  ```

  zur Funktion **foo** gesprungen und die Rücksprungadresse in Register `%2` hinterlegt.
- **Implementierung einer Funktion:** Die Implementierung beginnt mit dem Label der Funktion, gefolgt vom "Prologue", der die Rücksprungadresse auf den Stack legt. Die Funktion endet mit dem "Epilogue", in dem die Rücksprungadresse vom Stack geholt und zurückgesprungen wird. Nach diesem Rezept kann:

  ```
  fn foo()
  {
      // ...
  }
  ```

  umgeschrieben werden in:

  ```
  foo     subq    8,          %SP,        %SP
          movq    %2,         (%SP)

          // ...

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

Schritt 1: Kombination des Funktionspatterns mit dem Pattern für Kontrollstrukturen

Zur Vertiefung von Session 10 soll hier ein einfaches Programm mit zwei Funktionen geschrieben werden, das man in ABC wie folgt ausdrücken könnte:

```
@ <stdio.hdr>

fn msg()
{
    printf("hello\n");
}

fn main()
{
    msg();
}
```

Allerdings soll hier deutlich werden, wie der Compiler im Allgemeinen mit einem String-Literal in einem Programm umgeht. Es wird jeweils implizit ein globales Array mit internem Namen angelegt. Das kann man sich in diesem Fall wie folgt vorstellen:

```
@ <stdio.hdr>

global L0: array[] of char = "hello\n";

fn msg()
{
    printf(L0);
}

fn main()
{
    msg();
}
```

Die Funktion `printf` bekommt dann als Argument die Adresse des ersten Charakters (also ein Zeiger auf `'h'`). Außerdem soll in dieser Session erarbeitet werden, wie man Funktionen mit Parametern implementiert. Das Beispielprogramm sollte deshalb nicht `printf` verwenden, sondern nur `putchar` (was man bei unserer Architektur mit einer einfachen Instruktion realisieren kann). Das ABC-Beispiel kann dazu umgeschrieben werden in:

```
@ <stdio.hdr>

global L0: array[] of char = "hello\n";

fn msg()
{
    local p: -> char = L0;
    local ch: char;

    while ((ch = *p) != 0) {
        putchar(ch);
        ++p;
    }
}

fn main()
{
    msg();
}
```

Der Funktionskörper von `msg` entspricht dann dem Programm, das wir bei der Übung *ULM on Paper* analysiert haben.

Nach unserem Pattern kann man das Programm wie folgt in Assembler umschreiben:

```
        .data
.L0     .string "hello\n"

        .text
        .equ    SP,         1
        loadz   0,          %SP
        call    main,       %2
        halt    0

// function foo
foo:
        subq    8,          %SP,        %SP
        movq    %2,         (%SP)
        // begin of function body

        .equ    p,          2
        .equ    ch,         3
        loadz   .L0,        %p
.while:
        movzbq  (%p),       %ch
        subq    0,          %ch,        %ch
        jz      .done
        putc    %ch
        addq    1,          %p,         %p
        jmp     .while
.done:

        // end of function body
        movq    (%SP),      %2
        addq    8,          %SP,        %SP
        ret     %2

// function main
main:
        subq    8,          %SP,        %SP
        movq    %2,         (%SP)
        // begin of function body

        call    foo,        %2

        // end of function body
        movq    (%SP),      %2
        addq    8,          %SP,        %SP
        ret     %2
```

Alle Labels, die für die Umsetzung von Kontrollstrukturen dienen oder für intern generierte globale Variablen, beginnen hier übrigens mit einem Punkt (das entspricht dem, was ein Compiler generiert).

Hier kommen nun mehrere Dinge zusammen:
- Das Pattern für Funktionen
- Und für die Implementierung des Funktionskörpers von `msg` unter anderem das Pattern für Kontrollstrukturen.

Aufgaben
1. Tippe alle Programme ab und teste sie. Analysiere insbesondere die letzte ABC-Variante, da diese recht genau der Assembler-Implementierung entspricht.
2. Markiere (mit Farbe 1) im Assembler-Code die Stellen, die `fn msg() {/* … */}` und `fn main() {/* … */}` entsprechen, also jeweils den Prologue und Epilogue der Funktionen.
3. Markiere (mit Farbe 2) im Assembler-Code den Funktionsaufruf von `msg`.
4. Markiere (mit Farbe 3) den Code, der für den String-Literal erzeugt wurde.
5. Markiere (mit Farbe 4) den Code für den Funktionskörper von `msg`.

Schritt 2: Pattern für einen Funktionsparameter

Nun soll die Funktion `msg` so abgeändert werden, dass du ihr einen Zeiger auf einen String übergeben kannst. In ABC könnte man das Programm so schreiben:

```
@ <stdio.hdr>

fn msg(str: -> char)
{
    local p: -> char = str;
    local ch: char;

    while ((ch = *p) != 0) {
        putchar(ch);
        ++p;
    }
}

fn main()
{
    msg("hello!\n");
    msg("hello, again!\n");
}
```

Damit ist `msg` jetzt eine Print-Funktion für einen String. In der Assembler-Umsetzung kannst du die Übergabe eines Arguments beim Funktionsaufruf einbauen, indem du dort das Argument auf den Stack legst. Der Funktionsaufruf von `msg("hello!\n")` kann dann so umgesetzt werden:

```
        subq    8,          %SP,        %SP
        loadz   .L0,        %2
        movq    %2,         (%SP)
        call    foo,        %2
        addq    8,          %SP,        %SP
```

Die Größe einer Adresse ist 8 Byte, deshalb wird der Stack-Pointer erst "um 8 nach links verschoben", dann wird die Adresse `.L0` in das Register `%2` geladen und dessen Wert an die Spitze des Stacks geschrieben. Danach erfolgt der Funktionsaufruf. Nach der Rückkehr vom Funktionsaufruf wird das Argument wieder vom Stack entfernt, also der Zeiger "um 8 nach rechts verschoben".

In der Implementierung der Funktion kann im Funktionskörper das Argument vom Stack geladen werden. Dabei verlässt man aber die "Reine Lehre vom Stack". Innerhalb des Funktionskörpers ist der Parameter das zweitoberste Element auf dem Stack (hat also die Adresse `%SP + 8`) und auf das wird direkt zugegriffen. Jeder mir bekannte Assembler bietet Fetch-Instruktionen an, bei denen man bei der Adresse zusätzlich ein konstantes Offset (wird in diesem Kontext Displacement genannt) angeben kann. Bei unserer Architektur können zum Beispiel mit:

```
        movq    8(%SP),     %p
```

8 Bytes (ein Quad-Word) von der Adresse `%SP + 8` ins Register `%2` geladen werden.

Aufgabe

Passe dein Programm entsprechend an:
- Das Programm hat jetzt zwei String-Literale. Verwende für die zugehörigen Strings Labels wie `.L0` und `.L1`.
- Passe den Funktionsaufruf an.
- Im Funktionskörper soll jetzt der Parameter vom Stack geladen werden.

Schritt 3: Pattern für mehrere Funktionsparameter und einen Rückgabewert

Als Beispiele für Funktionen mit Parametern und einem Rückgabewert soll folgendes Beispiel dienen:

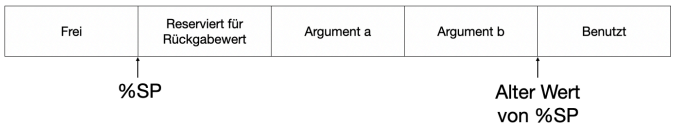
```
fn sum(a: u64, b: u64): u64
{
    return a + b;
}

fn main(): u64
{
    return sum(2, 3);
}
```

Beide Funktionen geben ein 64-Bit Integer zurück, wobei der Rückgabewert von main als Exit-Code verwendet wird (allerdings auf 8-Bit abgeschnitten). Ein mögliches Pattern dafür ist, dass der Aufrufer einer Funktion (der Caller) die Parameter auf dem Stack übergibt und auf dem Stack außerdem Platz für den Rückgabewert reserviert. Der Aufruf `sum(2,3)` kann dann so realisiert werden:

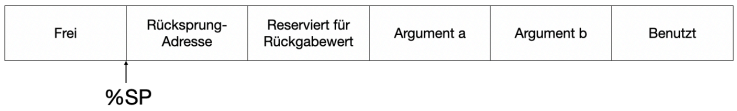
```
        subq    3*8,        %SP,        %SP
        loadz   2,          %2
        movq    %2,         8(%SP)
        loadz   3,          %2
        movq    %2,         16(%SP)
        call    sum,        %2
        movq    (%SP),      %3
        addq    3*8,        %SP,        %SP
```

Wenn die Call-Instruktion erreicht wird, kann man den Stack wie folgt beschreiben:



Nachdem die Funktion zurückkehrt, wird der Rückgabewert hier ins Register `%3` geschrieben.

In der aufgerufenen Funktion (dem Callee) "sum" kann der Stack, nachdem der Prologue ausgeführt wurde, so beschrieben werden:



Argument a liegt bei `%SP + 16`, Argument b liegt bei `%SP + 24`, der Rückgabewert sollte an die Adresse `%SP + 8` geschrieben werden.
Die Funktion kann also wie folgt implementiert werden:

```
// function sum
sum:
        subq    8,          %SP,        %SP
        movq    %2,         (%SP)
        // begin of function body

        .equ    a,          2
        .equ    b,          3
        movq    2*8(%SP),   %a
        movq    3*8(%SP),   %b
        addq    %a,         %b,         %a
        movq    %a,         1*8(%SP)

        // end of function body
        movq    (%SP),      %2
        addq    8,          %SP,        %SP
        ret     %2
```

Aber jetzt wird es wild: Die Funktion `main` hat ebenfalls einen Rückgabewert. Die Funktion wird nach der Initialisierung des Stacks aufgerufen. Und bei diesem Aufruf sollte dann ebenfalls Platz für den Rückgabewert reserviert werden:

```
        .equ    SP,         1
        loadz   0,          %SP

        subq    1*8,        %SP,        %SP
        call    main,       %2
        movq    (%SP),      %2
        halt    %2
```

Hier wird der Rückgabewert als Exit-Code verwendet. Da das Programm dann terminiert, muss der Stack-Pointer hier nach dem Funktionsaufruf nicht zurückgesetzt werden.

Die Funktion main ist also zunächst der Callee, wird dann zum Caller und wird dann wieder zum Callee, der zurückkehrt. Die Rollen **Caller** und **Callee** wechseln also.

Wie kann nun die Funktion main den von `sum` erhaltenen Rückgabewert selbst wieder zurückgeben? Das ist nun deine Aufgabe!

Schritt 4: Chill-out

In Session 10 habt ihr bereits Assembler-Code geschrieben, um von der Tastatur eine Zahl einzulesen. Den könnt ihr jetzt zum Abschluss in eine Funktion `readInt` packen und ein Programm wie dieses direkt in Assembler schreiben:

```
@ <stdio.hdr>

fn readInt(): u64
{
    local val: u64 = 0;
    local ch: int;

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

fn msg(str: -> char) {
    local ch: char;

    while ((ch = *str) != 0) {
        putchar(ch);
        ++str;
    }
}

fn main(): u64
{
    msg("Enter a number: ");
    local val: u64 = readInt();
    msg("Use echo $? and have a nice day!\n");
    return val;
}
```