LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 5
23.05.2022 - 29.05.2022

T5.1 Parsen mit `getopt`

In dieser Aufgabe wollen wir das Programm `numquad` um ein **C**ommand **L**ine **I**nterface (CLI) ergänzen. Hierzu werden wir die Funktion `getopt` verwenden, welche das Parsen der vom Benutzer übergebenen Argumente vereinfacht.

Vorlage: `https://gra.caps.in.tum.de/m/numquad.tar`

1. Bevor Sie mit der Implementierung des CLI beginnen, informieren Sie sich zunächst über die Funktion `getopt`. Nutzen Sie den Befehl `whatis getopt` (und evtl. `man man`) um herauszufinden, welche `man` page die relevanten Informationen der C-Funktion `getopt` beinhaltet. Lassen Sie sich diese anzeigen.

   Die `man` page `man 3 getopt` beinhaltet die relevanten Informationen.

2. Überschaffen Sie sich zunächst in den Abschnitten `NAME` und `SYNOPSIS` einen Überblick über die bereitgestellte Funktionalität und Signatur der `getopt` Funktion. Lesen Sie auch den ersten Absatz des Abschnitts `DESCRIPTION` um ein grobes Verständnis der Funktionsweise von `getopt` zu erlangen.

   `getopt` hat die Signatur `int getopt(int* argc, char * const argv[], const char* optstring)` und kann unter anderem Optionen der Form `-x` und `-y Y` parsen.

3. Welchen Header/Welche Header müssen Sie einbinden, um `getopt` verwenden zu können? Öffnen Sie die Datei `main.c` und fügen Sie diesen hinzu.

```c
...
#include <unistd.h>
...
```

4. Die `getopt` Parameter `argc` und `argv` entsprechen den bereits bekannten Parametern der `main` Funktion. Finden Sie die Bedeutung des `optstring` Parameters heraus.
   Hinweis: Mit `/optstring<ENTER>` können Sie in der `man` page nach dem Wort „optstring“ suchen. Weitere Suchergebnisse können Sie mit `n` (next) anzeigen lassen.

   `optstring` ist ein String, der die Namen und die Art der zu parsenden Programmargumente beschreibt.

5. Betrachten Sie die Usage- und Help-Messages am Anfang der Datei `main.c`. Formulieren Sie einen `optstring`, der das Parsen dieses CLI ermöglicht.

   Der `optstring` könnte z.B. wie folgt lauten: `"a:b:n:th"` (die Reihenfolge der Optionen ist hierbei egal). Die Optionen `-a`, `-b` und `-n` erwarten jeweils ein Argument, was mittels `:` im `optstring` spezifizierbar ist, die Optionen `-t` und `-h` hingegen nicht. Das Programmargument `fn` ist keine Option und wird anderweitig geparst.

6. Welche Rückgabewerte hat die Funktion `getopt` für den eben formulierten `optstring`? Betrachten Sie hierzu den Abschnitt `RETURN VALUE` der `man` page.

   Findet `getopt` in den Programmargumenten eine Option des `optstrings`, so gibt sie den Character-Wert dieser Option zurück. Wird hingegen eine Option gefunden, die nicht im `opstring` spezifiziert wurde, wird `'?'` zurückgegeben. `'?'` wird auch zurückgegeben, wenn `getopt` eine Option ohne Argument findet, diese Option aber ein Argument erwartet. Wurden alle Optionen behandelt, liefert `getopt` -1 zurück.

7. Integrieren Sie nun das Parsen der Optionen `-t` und `-h` in das Programm. Für andere Optionen soll die Programmausführung nach Ausgabe der Usage-Meldung zunächst abgebrochen werden.

```c
    ...
    int opt;
    while ((opt = getopt(argc, argv, "a:b:n:th")) != -1) {
        switch (opt) {
        case 't':
            return tests() ? EXIT_FAILURE : EXIT_SUCCESS;
        case 'h':
            print_help(progname);
            return EXIT_SUCCESS;
        default: /* '?' */
            print_usage(progname);
            return EXIT_FAILURE;
        }
    }
    ...
```

8. Die Optionen `-a`, `-b` und `-n` erwarten jeweils ein Argument. Finden Sie mithilfe des Abschnitts `DESCRIPTION` und/oder `EXAMPLE` der `man` page heraus, wie `getopt` diese beim Parsen zur Verfügung stellt.

   Findet `getopt` beim Scannen von `argv` eine Option, die ein Argument erwartet, wird die globale Variable `optarg` auf das folgende `argv` Element gesetzt.

9. Integrieren Sie nun auch das Parsen der verbleibenden Optionen. Für unsere Zwecke ist es ausreichend, `a` und `b` mit `atof` und `n` mit `atol`[1] zu konvertieren.

[1] Wir werden das robuste Konvertieren von Strings zu Integern auf einem zukünftigen Blatt noch genauer betrachten.

```c
    ...
    int opt;
    while ((opt = getopt(argc, argv, "a:b:n:th")) != -1) {
        switch (opt) {
        case 'a':
            a = atof(optarg);
            break;
        case 'b':
            b = atof(optarg);
            break;
        case 'n':
            n = atol(optarg);
            break;
        case 't':
            ...
```

10. Zuletzt wollen wir nun noch das Programmargument `fn` parsen. Finden Sie auch hierfür zunächst mithilfe der `man` page heraus, wie Sie darauf zugreifen können. Implementieren Sie basierend darauf die Parse-Funktionalität.

    Nach dem Parsen aller Optionen entspricht die globale Variable `optind` dem Index des ersten Elements in `argv`, das keine Option ist.

```c
    ...
    if (optind >= argc) {
        printf("%s: Missing positional argument -- 'fn'\n",
               progname);
        print_usage(progname);
        return EXIT_FAILURE;
    }

    const char* fn_name = argv[optind];
    ...
```

Änderungen an den `#includes`:

```c
...
#include <unistd.h>
...
```

Änderungen der `main` Funktion:

```c
    ...
    int opt;
    while((opt = getopt(argc, argv, "a:b:n:th")) != -1) {
        switch (opt) {
        case 'a':
            a = atof(optarg);
            break;
        case 'b':
```

```
                b = atof(optarg);
                break;
            case 'n':
                n = atol(optarg);
                break;
            case 't':
                return tests() ? EXIT_FAILURE : EXIT_SUCCESS;
            case 'h':
                print_help(progname);
                return EXIT_SUCCESS;
            default: /* '?' */
                print_usage(progname);
                return EXIT_FAILURE;
        }
    }

    if (optind == argc) {
        printf("%s: Missing positional argument -- 'fn'\n", progname);
        print_usage(progname);
        return EXIT_FAILURE;
    }

    const char* fn_name = argv[optind];
    ...
}
```

S5.1 Floating-Point Berechnungen

Bisher haben wir mit `add`, `sub` und `imul` einige Befehle für einfache Ganzzahlarithmetik kennen gelernt. Für die in vielen Anwendungen sehr wichtigen Fließkommaberechnungen funktionieren diese Instruktionen allerdings nicht.

Um nicht auf langsame softwareseitige Emulation der Gleitkommaarithmetik oder die x87-FPU zurückgreifen zu müssen, sind daher auf jedem x86-64 Prozessor die *Streaming SIMD Extensions (SSE)* verfügbar. Diese erweitern die Register um einen separaten Satz an 128-Bit großen SSE-Registern `xmm0`–`xmm15`, welche für Gleitkomma- und Vektoroperationen (Woche 6) verwendet werden können.

1. Finden Sie mithilfe der Intel-Dokumentation die Funktionsweise der folgenden Instruktionen heraus:

   `cvtsi2ss` `cvtss2si` `movss` `addss` `divss`

   Allgemeine Syntax wie gehabt: **Befehl Ziel,Quelle**

   - ***`cvtsi2ss`: "Convert Signed Integer to Scalar Single" - konvertiert eine Ganzzahl aus einem Standardregister oder dem Speicher in einen Single-Precision Floating-Point-Wert, der in ein SSE-Register geladen wird.***

- *`cvtss2si`: "Convert Scalar Single to Signed Integer" - konvertiert einen Single-Precision Floating-Point-Wert aus einem SSE-Register oder dem Speicher in eine Ganzzahl, die in ein Standardregister geladen wird. Der Wert wird dabei abgerundet.*
- *`movss`: "Move Single Scalar" - kopiert einen Single-Precision Floating-Point-Wert zwischen SSE-Registern und dem Speicher.*
- *`addss`: addiert einen Single-Precision Floating-Point-Wert aus einem SSE-Register oder dem Speicher auf einen Single-Precision Floating-Point-Wert in einem SSE-Register.*
- *`divss`: dividiert einen Single-Precision Floating-Point-Wert in einem SSE-Register durch einen Single-Precision Floating-Point-Wert aus einem SSE-Register oder dem Speicher. Die Operanden werden genau wie bei `addss` direkt angegeben, nicht wie bei `div`.*
- *`subss` und `mulss` funktionieren analog.*

2. Betrachten Sie nun exemplarisch das folgende Beispielprogramm. Was wird von der Funktion `func` berechnet? Wie wird die Ihnen bekannten Calling Convention in Bezug auf SSE-Register erweitert?

```c
#include <stdio.h>

float func(float x);
int main(int argc, char** argv) {
    float res = func(2.0);
    printf("Result: %f\n", res);
    return 0;
}
```

```asm
.intel_syntax noprefix
.global func
.text
func:
    mov r8,1
    cvtsi2ss xmm1,r8
    divss xmm1,xmm0
    movss xmm0,xmm1
    ret
```

Das Programm berechnet den reziproken Wert $1/x$ zu einer Zahl x. Im vorliegenden Fall wird die Zahl 0.5 ausgegeben.

Die Calling Convention wird wie folgt auf SSE-Register erweitert:

- *Rückgabewert: `xmm0`*
- *Argumente: `xmm0`–`xmm7`*
- *Scratch/temporär: `xmm0`–`xmm15`*

3. Wie müsste die Funktion verändert werden, sodass sie $\frac{1}{\sqrt{x}}$ berechnet? Welchen Befehl legt die Intel-Dokumentation nahe?

   Durch Einfügen des Befehls `sqrtss` wird

   $$y = \sqrt{\frac{1}{x}} = \frac{1}{\sqrt{x}}$$

   korrekt berechnet.

S5.2 Nutzung der SSE-Instruktionen

S5.2.1 Berechnung von Pi

Im Folgenden werden wir in x86-64-Assembler die Pi-Funktion

```
float pi(long n);
```

schreiben, deren Rückgabewert der Wert der folgenden Reihe ist:

$$\sum_{k=0}^{n} (-1)^k \frac{4}{2k+1}$$

Vorlage: `https://gra.caps.in.tum.de/m/pi.tar` – Für die Bearbeitung der Aufgabe müssen Sie lediglich die enthaltene Assembler-Datei bearbeiten.

1. Laden Sie zunächst die Konstante 4 in ein SSE-Register. Initialisieren Sie dann Ihr akkumulierendes SSE-Register sowie Ihren Schleifencounter.

   Aufgrund von Ungenauigkeiten bei der Rechnung mit Fließkommazahlen ist es empfehlenswert, die Schleife von $n \to 0$ iterieren zu lassen und nicht von $0 \to n$.

2. Sie werden eine Schleife benötigen, arbeiten Sie hierfür mit lokalen Labels. Sie können das unterste Bit des Schleifen-Zählers zur Entscheidung über Addition und Subtraktion verwenden. Inwiefern hilft Ihnen hierfür die Instruktion `test`?

   Die Instruktion `test` führt ein logisches „und" durch und setzt das Zero-Flag genau dann wenn das Ergebnis der Operation 0 ist. In diesem Fall also verwendet man also `test xX, 1`, um das unterste Bit zu prüfen. Dies kann dann mit `jz` oder `jnz` abgefragt werden.

Die Reihe konvergiert nur sehr langsam und Single-Precision-Fließkommazahlen haben eine relativ geringe Genauigkeit, weswegen selbst bei der gewählten Obergrenze von 10^6 das Ergebnis der Berechnung `3.141593` (echter Wert `3,1415926...`) noch ziemlich früh abweicht. Abgesehen davon sind Umwandlungen und Divisionen verhälnismäßig langsam; folgende Referenzlösung dient vornehmlich der Illustration der Verwendung dieser Instruktionen und ist nicht auf Performanz optimiert.

```
.intel_syntax noprefix
.global pi
.text
pi:
    // Note: cvtsi2ss is slow; loading 4.0 from
    // memory is likely to be faster.
    mov r8,4
    cvtsi2ss xmm3,r8         // xmm3 = 4.0
    xorps xmm0,xmm0          // xmm0 = sum = 0

.Lloop:
    lea rdx, [2*rdi + 1]     // rdx = 2k+1
    cvtsi2ss xmm2,rdx        // xmm2 = 2k+1
    movss xmm1,xmm3          // xmm1 = 4
    divss xmm1,xmm2          // xmm1 = 4/(2k+1)

    test rdi,1
    jnz .Lsub                // if k is odd subtract

    addss xmm0,xmm1          // xmm0 = sum + 4/(2k+1)
    jmp .Lcont

.Lsub:
    subss xmm0,xmm1          // xmm1 = sum - 4/(2k+1)

.Lcont:
    dec rdi                  // k--
    cmp rdi,0
    jge .Lloop               // loop while k >= 0
    ret
```

S5.2.2 Eulersche Zahl

Implementieren Sie eine Funktion, die den Wert der Eulerschen Zahl zurückgibt. Bedenken Sie, dass der Term $k!$ schnell sehr groß wird.

$$\sum_{k=0}^{n} \frac{1}{k!}$$

Unoptimierte Referenzlösung:

```
.intel_syntax noprefix
.global euler
.text
euler:
    mov r8,1
    cvtsi2ss xmm0,r8         // xmm0 = sum
    test rdi,rdi
    jz .Lend
    cvtsi2ss xmm2,r8         // xmm2 = 1/(k-1)!
    mov rcx,1                // rcx = k
```

```

.Lloop:
    cvtsi2ss xmm1,rcx        // xmm1 = k
    divss xmm2,xmm1          // xmm2 = (1/(k-1)!)/k = 1/k!
    addss xmm0,xmm2          // xmm0 = sum + 1/k!

    add rcx,1                // k++
    cmp rcx,rdi
    jle .Lloop               // loop while k <= n

.Lend:
    ret
```

P5.1 Kreisumfang

Es kommt regelmäßig vor, dass Floating-Point-Konstanten benötigt werden. Während es zwar theoretisch möglich ist, diese Konstanten immer neu zu berechnen, ist dies jedoch ineffizienter als das Vorberechnen vor der Ausführung. Im Folgenden werden wir betrachten, wie man Konstanten in der Assembler-Datei anlegen und verwenden kann.

Aufgabe: Schreiben Sie eine Funktion, die den Umfang eines Kreises mit einem gegebenen Radius r berechnet: $circ(r) = 2\pi r$

```
double circ(double radius);
```

1. Sie werden für die Implementierung dieser Funktion die Konstante 2π benötigen. Berechnen Sie diese, beispielsweise mit einem Taschenrechner.

   $2\pi = 6.2831853...$

2. Erinnern Sie sich an die Unterschiede der Datensektionen `.rodata`, `.data` und `.bss` (siehe Blatt 3). Welche Sektion ist für das Speichern von Konstanten am besten geeignet?

   Am besten geeignet ist die Sektion `.rodata`, alternativ ist technisch auch `.data` möglich (aber nicht empfohlen).

3. Mit der `.section` Direktive können Sie den Assembler anweisen, alles ab diesem Zeitpunkt stehende in eine Sektion mit dem angegebenen Namen zu schreiben. Fügen Sie (außerhalb einer Funktion) folgende Zeilen zu Ihrem Assembler-Code hinzu:

```
.section <sectionname>
// ...
.section .text
```

Finden Sie nun mithilfe der Referenz des GNU-Assemblers[2] heraus, mit welcher Direktive Sie `float`/`double`-Werte generieren können. Fügen Sie die entsprechende Direktive mit der oben berechneten Konstante an der passenden Stelle ein und versehen Sie diese mit einem geeigneten Label.

Eine unvollständige Auflistung von Direktiven um Konstanten zu schreiben:

- *`.float` **für 32-Bit Floating-Point Zahlen***
- *`.double` **für 64-Bit Floating-Point Zahlen***
- *`.byte` **für 8-Bit Integer***
- *`.2byte` **für 16-Bit Integer***
- *`.4byte` **für 32-Bit Integer***
- *`.8byte` **für 64-Bit Integer***
- *`.asciz` **für C-Strings (NUL-terminiert)***

```
  .section .rodata
.Ltau:
  .double 6.2831853
  .section .text
```

4. Mit folgendem Speicheroperanden können Sie nun auf die Zahl zugreifen, z.B. via:

```
  movsd xmm1, [rip + .Ltau]
```

Vervollständigen Sie nun die Implementierung Ihrer Funktion.

Hinweis: Sie benötigen einschließlich `ret` nur 2 Instruktionen.

Vollständige Referenzlösung:

```
  .section .rodata
.Ltau:
  .double 6.2831853
  .section .text
  .global circ
circ:
  mulsd xmm0, [rip + .Ltau]
  ret
```

P5.2 Numerische Quadratur [4 Pkt.]

Für eine gegebene Funktion $f : \mathbb{R} \to \mathbb{R}$ soll auf dem Intervall $[a; b]$ das Integral approximiert werden. f soll in dem Intervall an n gleichmäßig verteilten Stützstellen evaluiert

[2] `https://sourceware.org/binutils/docs/as/`

werden. Zwischen den Stützstellen soll linear interpoliert werden. Im Fall $n = 2$ wird die Funktion also genau an den Stellen a und b evaluiert und die Fläche des Trapezes ist:

$$\frac{(b-a) \cdot (f(b) + f(a))}{2}$$

Bei $n = 3$ sind die Stützstellen folglich $\{a, \frac{a+b}{2}, b\}$ und es wird die Fläche von zwei Trapezen berechnet. Für $n < 2$ ist das Ergebnis undefiniert, dies soll in der Implementierung durch den Wert *NaN* dargestellt werden.

Aufgabe: Implementieren Sie folgende Funktion in Assembler:

```
double numquad(double(* f)(double), double a, double b, size_t n);
```

Hinweis: Achten Sie auf die Calling Convention, sowohl in Bezug auf die aufrufende Funktion *als auch* im Bezug auf die Funktion `f`, die von Ihrer Implementierung aufgerufen wird. Beachten Sie hierbei insbesondere das Stack-Alignment und Caller-saved Register.

Vorlage: `https://gra.caps.in.tum.de/m/numquad.tar` – falls Sie T5.1 nicht bearbeitet haben, heben Sie einfach die Kommentierung der annotierten Zeile in der `main` Funktion auf.

1. Berechnen Sie zunächst die Schrittweite zwischen den Stützstellen und behandeln Sie den Fall, dass $n < 2$ gilt.
$$s = \frac{b-a}{n-1}$$

2. Iterieren Sie nun in einer Schleife über die Trapeze zwischen den Stützstellen. Addieren Sie dabei in jeder Iteration die zuvor berechnete Schrittweite auf den aktuelle "linke" Ende des Trapezes. Summieren Sie die Flächen der Trapeze auf und geben Sie den Wert am Ende zurück.

   Es empfiehlt sich, zunächst eine einfache Funktion wie x^2 direkt in den Code zu schreiben und die übergebene Funktion noch nicht aufzurufen. Stellen Sie zunächst sicher, dass Ihr Code für diesen Fall korrekte Ergebnisse liefert.

3. Passen Sie Ihre Funktion nun so an, dass die übergebene Funktion (ein Funktionspointer) mit einem indirekten Funktionsaufruf über `call` aufgerufen wird. Beachten Sie hierbei folgende Punkte:

   - Die Funktion darf alle *caller-saved* Register beliebig überschreiben. Nutzen Sie daher *callee-saved* Register, um Werte zwischenzuspeichen. Speichern Sie insbesondere auch den Funktionspointer selbst in einem *callee-saved* Register.

     Hinweis: Nutzen Sie nicht `push`/`pop` unmittelbar vor/nach dem Funktionsaufruf – dies führt zu vielen unnötigen Speicherzugriffen.

   - Alle SSE-Register sind *caller-saved*.

- Der Stack-Pointer muss zum Zeitpunkt *vor* dem `call` ein Alignment von 16-Byte aufweisen. Falls dies nicht so sein sollte, kann es zu einem *Segmentation Fault* kommen. Bedenken Sie auch, dass zu Beginn Ihrer Funktion das notwendige Stack-Alignment nicht gegeben ist, da `call` bereits 8-Byte auf den Stack gelegt hat.

Stellen Sie sicher, dass Ihre Implementierung auf den in den Materialien bereitgestellen Testfällen funktioniert.

4. Optimieren Sie Ihre Funktion, indem Sie redundante Funktionsaufrufe mit demselben Parameter eliminieren.

$$
\begin{aligned}
numquad(f,a,s,n) &= \sum_{i=0}^{n-2} \frac{s\cdot(f(a+i\cdot s)+f(a+(i+1)\cdot s))}{2} \\
&= s\cdot\sum_{i=0}^{n-2} \frac{f(a+i\cdot s)+f(a+(i+1)\cdot s)}{2} \\
&= s\cdot\left(\frac{f(a)}{2}+\sum_{i=1}^{n-2}\frac{f(a+i\cdot s)+f(a+i\cdot s)}{2}+\frac{f(a+(n-1)\cdot s)}{2}\right) \\
&= s\cdot\left(\frac{f(a)}{2}+\sum_{i=1}^{n-2} f(a+i\cdot s)+\frac{f(b)}{2}\right) \\
&= s\cdot\left(\sum_{i=0}^{n-1} f(a+i\cdot s)-\frac{f(a)+f(b)}{2}\right)
\end{aligned}
$$

Referenzlösung:

```
numquad:
  cmp rsi, 2
  jb .LNaN

  push rbx
  push rbp
  sub rsp, 0x28
  // Stack frame layout: {0x0: s, 0x8: f(a), 0x10: acc, 0x18: cur}

  mov rbx, rdi              // rbx = f
  lea rbp, [rsi - 1]        // loop counter
  // Compute step width s = (b-a)/(s-1)
  movsd xmm2, xmm1
  cvtsi2sd xmm3, rbp
  subsd xmm2, xmm0
  divsd xmm2, xmm3
  movsd [rsp], xmm2         // store s
  movsd [rsp+0x18], xmm0    // store cur = a

  call rbx                  // f(a)
  movsd [rsp + 0x8], xmm0   // store f(a)
  movsd [rsp + 0x10], xmm0 // initialize acc

```

```
.Lloop:
  movsd xmm0, [rsp + 0x18]
  addsd xmm0, [rsp]
  movsd [rsp + 0x18], xmm0 // update cur += s
  call rbx                 // f(cur)
  sub rbp, 1
  movsd xmm1, [rsp + 0x10]
  addsd xmm1, xmm0
  movsd [rsp + 0x10], xmm1 // update acc += f(cur)
  jnz .Lloop

  addsd xmm0, [rsp + 0x8]
  mulsd xmm0, [rip + .LChalf] //      (f(a) + f(b)) * -0.5
  addsd xmm0, [rsp + 0x10] //   acc + (f(a) + f(b)) * -0.5
  mulsd xmm0, [rsp]        // (acc + (f(a) + f(b)) * -0.5) * s
  add rsp, 0x28
  pop rbp
  pop rbx
  ret

.LNaN:
  movsd xmm0, [rip + .LCNaN]
  ret

  .section .rodata
  .align 8
.LCNaN:  .8byte 0xFFF8000000000000
.LChalf: .double -0.5
```

P5.3 Round [2 Pkt.]

In dieser Aufgabe soll in C eine Funktion implementiert werden, welche eine rationale Zahl zur einer ganzen Zahl rundet. Die Eingabezahl ist dabei entweder eine Double-Precision Floating-Point Zahl oder eine 32.32-Bit Fixed-Point Zahl; gerundet soll entweder zur nächst größeren oder zur nächst kleineren ganzen Zahl. Das Ergebnis soll in gleiche Format wie die Eingabe zurückgegeben werden.

Eine Zahl wird durch folgende Datenstruktur dargestellt. Ob eine Zahl als Floating-Point-Zahl dargestellt wird, wird durch `isflt` angegeben. Innerhalb der `union`[3] wird die Fixed-Point-Zahl als `int64_t` oder die Floating-Point-Zahl als `double` gespeichert.

```
struct num { bool isflt; union { int64_t fix; double flt; }; };
```

Die Funktion hat folgende Signatur:

[3]Eine `union` ist wie ein `struct`, mit dem Unterschied, dass alle Datentypen `an derselben Adresse` beginnen. Man kann damit also nur eines der Felder gleichzeitig sinnvoll nutzen. Üblicherweise wird ein einem `struct`-Member vor der `union` angezeigt, welches der Felder Gültigkeit hat, hier durch `isflt`. In diesem Fall handelt sich um eine *anonyme* `union`, auf deren Member beispielsweise über `num.fix` zugegriffen werden kann.

```
enum RoundMode { RM_FLOOR = 1, RM_CEIL };
struct num num_round(struct num num, enum RoundMode rm);
```

1. Behandeln Sie zunächst den Fall von Fixed-Point-Zahlen. Beachten Sie, dass es sich um eine vorzeichenbehaftete Zahl handelt.

2. Um Floating-Point-Zahlen zu behandeln, wandeln Sie diese zunächst bitweise in einen 64-Bit Integer um.

   Hinweis: Verwenden Sie hierzu eine `union`; ein Pointer-Cast zu einem anderen Datentyp ist *Undefined Behavior*.

   Folgende Makros könnten zu diesem Zweck verwendet werden:

```c
#define I64_AS_F64(v) ((union { int64_t i; double d; }) { .i = v }.d)
#define F64_AS_I64(v) ((union { int64_t i; double d; }) { .d = v }.i)
```

3. Behandeln Sie nun den Fall, dass der Exponent kleiner als 0 ist. Welchen maximalen Wert kann die Repräsentation der Floating-Point-Zahl in diesem Fall annehmen? Welche mögliche Ergebnisse kann die Rundungsoperation daher haben? Denken Sie auch an das Vorzeichen.

4. Anschließend sollten Sie den Fall betrachten, dass aufgrund des Exponenten die Floating-Point-Zahl nicht notwendigerweise eine ganze Zahl ist. Nutzen Sie zur Rundung der Mantisse dieselbe Methodik wie bei Fixed-Point-Zahlen. Beachten Sie, dass Sie möglicherweise auch den Exponenten anpassen müssen.

5. Warum ist für größere Exponenten keine explizite Behandlung notwendig? Achten Sie darauf, dass Werte wie *NaN* oder `Infinity` unverändert bleiben sollen.

Referenzlösung:

```c
struct num num_round(struct num num, enum RoundMode rm) {
  if (num.isflt) {
    // Well, there is no need for an explicit conversion:
    // we already have a suitable union for this purpose.
    int64_t vi = num.fix;
    int exp = ((vi >> 52) & 0x7ff) - 0x3ff;
    if (exp < 0) {
      if (rm == RM_FLOOR)
        num.flt = vi == INT64_MIN ? -0.0 : vi < 0 ? -1.0 : 0.0;
      else
        num.flt = vi < 0 ? -0.0 : vi == 0 ? 0.0 : 1.0;
    } else if (exp < 52) {
      int64_t msk = 0xfffffffffffff >> exp;
      if (rm == RM_FLOOR)
        num.fix = (vi + (vi < 0 ? msk : 0)) & ~msk;
      else
        num.fix = (vi + (vi >= 0 ? msk : 0)) & ~msk;
    } // else -- already integral, Inf, or Nan => no change
  }
  else {
    int64_t msk = 0xffffffff;
    num.fix = (num.fix + (msk * (rm == RM_CEIL))) & ~msk;
  }
  return num;
}
```

Q5.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)