Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

3. Aufgabenblatt mit Lösungsvorschlag

08.05.2023
Konzepte der maschinennahen Programmierung, C-Programmierung

Aufgabe 1: Theoriefragen

(a) Wie kann mit Maschinenbefehlen die Multiplikation mit 2 und die Division mit 2 effizient implementiert werden? Wie heißen diese Maschinenbefehle bei einer ARM-Architektur?

(b) Mittels welcher Flags werden Verzweigungen im Programmablauf realisiert?

Lösungsvorschlag:

(a) Durch das Schieben nach links, ergibt sich eine Multiplikation mit 2. Entsprechend wird eine Division durch 2 durch das Schieben nach rechts realisiert. Die Maschinenbefehle heißen lsl (für den Links-Shift) und lsr (für den Rechts-Shift).

(b) Die Flags heißen:

- C – Carryflag (Übertragsflag)
- Z – Zeroflag (Nullflag)
- N – Negativflag (Vorzeichenflag)
- V – Overflowflag (Überlaufflag)
- Bemerkung: beim ARM-Prozessor werden die Flags im CPSR - Current Program Status Register abgespeichert und werden dort mit C, Z, N, V bezeichnet.

Aufgabe 2: Implementierung von Hochsprachenkonstrukten in Assembler

Übersetzen Sie folgende Hochsprachenkonstruke in ARM-Assembler. Kommentieren Sie Ihren Assemblercode.

(a)
```
if (a == o)
  f = i + 1;
else
f = f - i;
```

Geben Sie das Assemblerprogramm ein und testen Sie das Programm mit den folgenden Werten:

- a = 1
- o = 2
- f = 5
- i = 1

Als Rahmen können Sie das Programm aus der 2. Übung verwenden. Schreiben Sie f als Rückgabewert in das Register r0.

(b)
```
int pow = 1;
int x = 0;
while (pow != 256) {
pow = pow * 2;
x = x + 1;}
```

Lösungsvorschlag:

(a)
```
/* -- if.s */
/* Kommentar  */
.global main /* Definition des sogenannten Einsprungpunktes fuer das Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0,#1  /* r0 = a = 1 */
    mov r1,#2  /* r1 = o = 2 */
    mov r2,#5  /* r2 = f = 5 */
    mov r3,#1  /* r3 = i = 1 */
    cmp r0,r1
    bne L1
    add r2,r3,#1
    b L2
    L1:
    sub r2,r2,r3
    L2:
    mov r0,r2
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

(b)
```
/* -- Schleife.s */
/* Kommentar  */
.global main /* Definition des sogenannten Einsprungpunktes fuer das Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0,#1  /* r0 = pow = 1 */
    mov r1,#0  /* r1 = x = 0 */
    WHILE:
    cmp r0,#256
    beq DONE
    lsl r0,r0,#1 /* pow = pow * 2 */
    add r1,r1,#1 /* x = x + 1 */
    b WHILE
    DONE:
    mov r0,r1
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

Aufgabe 3: Makefiles

Hinweis: Makefiles sind nicht klausurrelevant, erlauben aber ein effizienteres Arbeiten bei den weiteren Übungen.

Damit die Befehle zum Kompilieren nicht jedes mal erneut eingegeben werden müssen, gibt es eine einfache Methode, die sich Makefiles nennt. Hier muss zum Assemblieren und Kompilieren nur ein einfacher Befehl in die Kommandozeile eingegeben werden. Diejenigen Befehle, die sonst jedes mal erneut einzeln eingegeben werden müssten, werden in eine Datei geschrieben.

Es funktioniert folgendermaßen:

1. Erstellen Sie eine Datei mit dem Namen $Makefile$
2. Schreiben Sie in die erste Zeile all :

3. Schreiben Sie in die darauf folgenden Zeilen die Shellbefehle, die Sie ausführen wollen.
4. Führen Sie in der Shell den Befehl *make* aus, dadurch werden die einzelnen Befehle welche sich in dem Makefile befinden, ausgeführt.

Zur Erinnerung: die Befehle zum Assemblieren und Linken eines Assemblerprogramms `example.s` sind

```
arm-linux-gnueabihf-as -o example.o example.s
```

```
arm-linux-gnueabihf-gcc -o example example.o
```

Dies war nur eine kurze Einführung in Makefiles. Ausführlichere Tutorials gibt es in den Materialien auf Moodle und an verschiedenen Stellen im Internet.

Lösungsvorschlag:

Ein einfaches Makefile kann folgendermaßen aussehen:

```
all:
    arm-linux-gnueabihf-as -o example.o example.s
    arm-linux-gnueabihf-gcc -o example example.o
```

Aufgabe 4: Einführung in C

Die Programmiersprache C wurde 1969 in den Bell Laboratories entwickelt. Wie so oft führte Unzufriedenheit mit einer Sprache, in diesem Fall B(CPL)[1] zur Entwicklung der Sprache C. Die Entwickler waren Dennis Ritchie und Ken Thompson. Letzterer war auch an der Entwicklung des Betriebssystems Unix beteiligt. In den 70er Jahren gab es eine Vielzahl von C-Dialekten. Die Lösung war die Schaffung eines C-Standards. Von Brian W. Kernighan und Dennis Ritchie wurde 1978 das Buch „The C Programming Language“ veröffentlicht. Daraus wurde der K & R Standard entwickelt, der dann im ANSI[2] C überführt wurde. Warum nutzt man heute noch die Sprache C?

- Die Sprache ist einfach aufgebaut – Grundlage für viele andere Sprachen (in Syntax und Semantik)
- Im Bereich der Eingebetteten Systeme und Unix/Linux hohe Verbreitung
- Maschinennahe Programmierung einfach möglich – z. B. Schreiben von Inline-Assembler-Code

Eine kurze Einführung in C sowie eine Literaturliste ist im Foliensatz Eine kurze Einführung in die Programmiersprache C zu finden.

Im Folgenden das Standardprogramm, welches in jeder Programmiersprache zum Testen verwendet wird.

```
#include<stdio.h>
int main(){
printf("Hello World\n");
return 0;
}
```

Zum Übersetzen eines C Programms soll im Folgenden der GCC Compiler benutzt werden. Die Übersetzung der Datei mit dem Namen hello.c wird durch folgende Befehle erreicht. Achtung: sh$ muss nicht eingegeben werden.

```
sh$ gcc -o hello hello.c        (compilieren)
sh$ ./hello                     (Ausführen des Executables)
```

Der optionale Parameter -o mit der folgenden Zeichenkette hello gibt den Namen der übersetzten (und ausführbaren) Datei an.

Gegeben ist folgendes C-Programm:

[1] Basic Combined Programming Language
[2] American National Standard Institute

```c
/* What-Do-I-Do */
#include<stdio.h>

int main(){

short int i = 1;
short int n = 32767;
for (i;i<=n;i++)
printf("i␣ist␣%i␣\n",i);
return 0;
}
```

Was passiert bei der Ausführung des Programms?

Lösungsvorschlag:

Die Ausgabe fängt bei $i = 0$ an und geht bis $i = 32767$, danach entsteht ein Überlauf, da die höchste positive darstellbare Zahl bei short int 32767 liegt. Die nächste Zahl (vgl. Darstellung der Zweikomplementzahlen im Zahlenkreis) ist die Zahl - 32768. Diese Zahl ist wiederum kleiner als n (Abfrage im C-Programm i<=n). Da diese Bedingung nicht erfüllt werden kann, liegt hier eine Endlosschleife vor und das Programm terminiert nicht.

Aufgabe 5: Fehlersuche

Übersetzen Sie das folgende C-Programm in ARM-Assembler und führen Sie es aus. Prüfen Sie anschließend dessen Rückgabe mit `echo $?`. Entspricht das ausgegebene Ergebnis Ihren Erwartungen? Worin könnten eventuelle Unstimmigkeiten begründet sein?

```c
int main() {
   int a = 53;
   int b = a << 4;
   int c = b + 5;
   return c;
}
```

Lösungsvorschlag:

```
.global main

main:
    mov r0, #53     /* r0 <- a = 53 */
    lsl r1, r0, #4  /* r1 <- b = a << 4 */
    add r2, r1, #5  /* r2 <- c = b + 5 */
    mov r0, r2      /* r0 <- c */
    bx lr           /* return c */
```

Es ist zu beobachten, dass das Ergebnis der Berechnung bei der Ausführung nicht korrekt ausgegeben wird. Das Problem ist darin begründet, dass das Ergebnis der Berechnung $(53 * 2^4) + 5 = 853$ nicht als Fehlercode dargestellt werden kann. Fehlercodes können auf POSIX-Systemen nur Werte im Bereich von 0 bis 255 annehmen[3]. Dabei signalisiert der Wert 0 eine erfolgreiche Ausführung des Programmes und die Werte 1 bis 255 einen Fehler bei der Ausführung des Programmes.

Aufgabe 6: Monte-Carlo-Simulation

Monte-Carlo-Simulationen können zur Berechnung des Inhalts regel- und unregelmäßiger Flächen und Körper verwendet werden. Die Algorithmen werden als Monte-Carlo-Algorithmen bezeichnet und gehören in die Klasse der

[3] https://shapeshed.com/unix-exit-codes/

randomisierten Algorithmen. Das Verfahren der Monte-Carlo-Simulation soll anhand der probabilistischen Bestimmung der Kreiszahl π verdeutlicht werden. Hierzu werden zufällige Punkte $\{(x,y)|x \in [-1,1], y \in [-1,1]\}$ generiert und es wird überprüft, ob diese innerhalb des Einheitskreises liegen. Die sich ergebende Wahrscheinlichkeitsverteilung erlaubt die Berechnung der Kreiszahl π nach folgender Formel.

$$\frac{\text{Treffer in Kreisflaeche}}{\text{generierte Punkte im Rechteck}} = \frac{r^2 \cdot \pi}{(2 \cdot r)^2} = \frac{\pi}{4}$$

a) Schreiben Sie ein C Programm, welches die Kreiszahl π berechnet und ausgibt. Die Funktion rand() liefert Zufallszahlen.[4]

Lösungsvorschlag:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
  const long n = 200000000;
  srand((unsigned int) time(0)); // set random seed

  long count = 0;
  for (long i = 0; i < n; ++i) {
    // Generate points on quad. (x,y) should be in [0,1]
    const double x = rand() / (double) RAND_MAX;
    const double y = rand() / (double) RAND_MAX;
    // Check whether points are in circle
    if (x*x + y*y <= 1) { ++count; }
  }

  const double pi = 4.0*count / (double) n;
  printf("Die Kreiszahl pi ist: %f\n", pi);
 return 0;
}
```

b) Testen Sie Ihr Programm mit unterschiedlicher Anzahl von Punkten (im Bereich von ca. 100000 bis 100000000). Notieren Sie die Anzahl der Punkte und die Genauigkeit in einer Tabelle.

Lösungsvorschlag:

Die Tabelle stellt jeweils zwei Ausführungen des Programms dar.

	1. Versuch		2. Versuch	
Anzahl der Tests	Wert	Fehler	Wert	Fehler
100000	3.138440	-0.003152	3.143960	0.002367
1000000	3.143972	0.002379	3.143204	0.001611
10000000	3.141396	-0.000196	3.141780	0.000187
100000000	3.141675	0.000082	3.141606	0.000013

c) Die Berechnung von π kann auch auf ein Flächenintegral zurückgeführt werden.

$$\int_0^1 \frac{4}{1+x^2} dx$$

[4] Binden Sie falls notwendig die entsprechenden Librarys ein. Achten Sie bei rand() darauf vorher einen zufälligen Seed zu setzen.

Schreiben Sie ein C Programm, welches die Integration (z. B. Trapezintegration) ausführt und vergleichen Sie die Ergebnisse mit b).

Lösungsvorschlag:

```
#include <stdio.h>
#include <math.h>

static long num_steps = 1000000;
double step;

int main()
{
  long i;
  double x, pi, sum = 0.0;

  step = 1.0 / (double) num_steps;
  for (i = 1; i <= num_steps; i++)
  {
    x = (i - 0.5) * step;
    sum = sum + 4.0 / (1.0 + x * x);
  }

  pi = step * sum;
  printf("PI %lf\n", pi);
 return 0;
}
```

Schrittweite	Wert	Fehler
1	3.200000	0.058407
10	3.142426	0.000833
100	3.141601	0.000008
1000	3.141593	0.000000

Der Vergleich der Tabellen zeigt, dass das Ergebnis bei der Integration deutlich schneller zum „richtigen“ Ergebnis führt.