Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

2. Aufgabenblatt

24.4.2023
Einführung in die maschinennahe Programmierung

Aufgabe 1: Theoriefragen

(a) Erklären Sie den Unterschied zwischen Big-Endian und Little-Endian.

(b) Wann sollten Daten in Registern gespeichert werden?

(c) Gegeben sei ein wort-adressierter Speicher. Wie viele Wörter lassen sich maximal adressieren, wenn die Adresse zwei Byte groß ist?

Aufgabe 2: Grundlegende Maschinenbefehle der ARM Architektur

In der Vorlesung haben Sie den Befehl add kennengelernt. Zur Erinnerung: der Ausdruck, der in einer Hochsprache als $a = b+c;$ geschrieben wird, wird in Assemblercode zu add r0,r1,r2. Dabei wird davon ausgegangen, dass $r0 = a, r1 = b$ und $r2 = c$ ist.

Neben Additionsbefehlen gibt es eine Vielzahl von Befehlen, die der Programmiererin/dem Programmierer zur Verfügung stehen.

- Subtraktionsbefehl: sub
- Transportbefehl: mov
- ...

Wandeln Sie folgenden Hochsprachen-Code in ARM-Assemblercode um. Die Variablen a, b, c sind in den Registern r0, r1, r2 abgelegt.
Hinweise:

- Multiplikationen kann man als Additionen formulieren.
- ARM besitzt einen Additionsbefehl, der die Angabe von Konstanten im Befehl erlaubt.

(a) $c = 20$

(b) $b = b * 4$

(c) $a = c + 42$

(d) $a = b * 2 + c + 1$

(e) $a = a * 3$

Aufgabe 3: ARM reversed

Wandeln Sie folgenden ARM-Assemblercode jeweils in eine einzige Zuweisung in Hochsprachen-Code um. Die Variable a liegt in r0, Kopien der Variablen b und c liegen in r1 bzw. r2.

(a)
```
add r0, r0, #2
add r0, r0, r0
add r0, r0, r2
```

(b)
```
add r1, r1, r1
add r2, r2, r2
add r2, r2, #1
add r2, r2, r1
add r1, r1, r1
add r1, r1, #42
add r0, r1, r2
```

Aufgabe 4: Programmierung in Assembler

In der Vorlesung wurde das Assemblerprogramm vom C-Compiler erzeugt. Im Folgenden soll Assemblercode von Hand geschrieben werden. Da einige grundlegende Ideen (z. B. die Unterprogrammtechnik) erst später behandelt werden, werden die Ergebnisse der Programme als sogenannter Fehlercode an die Shell (vgl. Übung 1) zurückgegeben.

Den ARM-Server erreichen Sie unter der Adresse `TU-ID@clientssh-arm.rbg.informatik.tu-darmstadt.de`. Zur Erinnerung: Die ISP betreibt sogenannte Login-Knoten. Der Login erfolgt über SSH[1]. Eine Einleitung zur Einrichtung finden Sie unter `https://support.rbg.informatik.tu-darmstadt.de/wiki/de/doku/computerhilfe/ssh`.

Geben Sie folgendes Assemblerprogramm ein.

```
/* -- first.s */
/* Kommentar  */
.global main /* Definition des sogenannten Einsprungpunktes fuer das Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0, #42 /* Schreibe eine 42 in das Register r0 */
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

Nach Eingabe des Programms (vgl. Übung 1) muss das Programm assembliert und gelinkt werden. Dies geschieht beim ARM-Server clientssh-arm durch folgende zwei Kommandos[2].

[1] Secure Shell

[2] Raspberry Pi Nutzer bitte „Zum Weiterlesen“ springen.

1. `arm-linux-gnueabihf-as -o first.o first.s`
2. `arm-linux-gnueabihf-gcc -o first first.o`

Der erste Befehl ruft den Assembler auf und erzeugt ein sogenanntes Objektfile (vgl. Vorlesung 2). Durch den Parameter -o kann man den Namen des Ausgabefiles angeben. Der zweite Befehl übersetzt das Objektfile in ein ausführbares File.

Damit der Fehlercode, der in diesem Fall das Ergebnis unserer Zuweisung ist, ausgegeben wird, muss das Programm wie folgt aufgerufen werden: `./first ; echo $?`

Tipp: Mit den Cursortasten lassen sich auf der Shell die eingegebenen Kommandos wieder zurückholen und erneut ausführen. Das Schreiben eines kleinen Skripts erleichtert ggf. den Übersetzungsvorgang.

Formulieren Sie nun das folgende C-Programm händisch in ein ARM-Assemblerprogramm, übersetzen und linken Sie das Programm und führen es aus.

```
/* Addition */
int main(){

int p = 5;
int q = 12;
int result = p + q;
return result;
}
```

Zum Weiterlesen:

Die verfügbaren Assembler, Compiler und Linker hängen vom Betriebssystem ab. Auch die Frage, ob Assembler, Compiler und Linker 32 Bit oder 64 Bit Code übersetzen können, hängt vom installierten Betriebssystem (32 Bit/ 64 Bit) ab[3]. Bei der Ausführung der Programme ist ebenfalls die Frage nach 32 Bit und 64 Bit Maschinencode von Interesse[4]. Es gibt eine Anzahl von 64 Bit Prozessoren (z. B. Raspberry Pi 3 Modell B+) die mit einem 32 Bit Betriebssystem (z. B. Raspberry Pi OS) laufen. Die Umstellung des Betriebssystems auf 64 Bit erfolgt relativ langsam. Die Gründe dafür sind:

- Viele Raspberry Pis haben max. 1 GB Speicher. Von dem erweiterten Adressraum (vgl. Vorlesung Betriebssysteme) wird man kaum profitieren. Neue Modelle haben bis zu 8 GB Speicher. Gleichwohl ist der Punkt Wiederverwendbarkeit in der Maker-Szene ein wichtiger.
- Ein Gewinn an Ausführungsgeschwindigkeit durch die 64 Bit-Maschinenbefehle ist bei den typischen Anwendungen eines Raspberry Pis nicht zu erwarten.
- ...

Die Übersetzung der oben aufgeführten Programme auf dem Raspberry Pi mit dem Betriebssystem Raspberry Pi OS erfolgt durch folgende zwei Kommandos.

- `as -o first.o first.s`
- `gcc -o first first.o`

[3] Das Thema Cross-Compiling wird noch behandelt.

[4] Hinweis: nicht jeder ARM-Prozessor kann 32 Bit Befehle ausführen.