Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

7. Aufgabenblatt mit Lösungsvorschlag

12.6.2023
Compilieren, Assemblieren und Linken

Aufgabe 1: Maschinensprache vs. Hochsprache

a) Wann ist es sinnvoll, Programme direkt in Assembler (statt in C oder JAVA) zu schreiben?

Lösungsvorschlag:

- der Speicher knapp ist
- Programmierung von Echtzeitsysteme mit kritischen Antwortszeiten
- Handoptimierungen aus Performanzgründen notwendig sind
- direkter Zugriff auf den Prozessor nötig ist

b) Nennen Sie einige Vorteile einer Hochsprache wie C gegenüber Assembler.

Lösungsvorschlag:

- Kontrollstrukturen sind vorhanden
- Besseres Verständnis des Codes
- Datenstrukturen sind vorhanden
- Type-Checking
- Bessere Wartbarkeit des Codes

Aufgabe 2: Parity-Bit

In der Netzwerktechnik werden oft verschiedene Arten von Parity-Bits genutzt, um sicher zu gehen, dass die Datenübertragung einwandfrei funktioniert hat. Das Parity-Bit in seiner einfachsten Form ist die Vervollständigung einer Bitfolge auf eine gerade Anzahl von 1er durch anfügen eines Bits. Als Beispiel wird die Bitfolge *1011* auf *1011 1* und *0110* auf *0110 0* erweitert. Schreiben Sie ein ARM-Assemblerprogramm, das eine gegebene 4-Bit Zahl in `r0` um das oben beschriebene Parity-Bit erweitert und das Ergebnis wieder in `r0` ablegt. Sprünge oder konditionale Befehle stehen dabei nicht zur Verfügung. Kommentieren Sie Ihren Code. Lösungsvorschlag:

```
/* -- parity.s */
/* Kommentar  */
.global main /* Einsprungpunkt Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0,#13 /* 13 = 0b1101 */
    mov r2,#0  /* Ergebnisregister 0 */
    /* 000x zaehlen */
    mov r1,r0
    and r1,r1,#1
    add r2,r2,r1
    /* 00x0 zaehlen */
    mov r1,r0
    and r1,r1,#2
    lsr r1,r1,#1
    add r2,r2,r1
    /* 0x00 zaehlen */
    mov r1,r0
    and r1,r1,#4
    lsr r1,r1,#2
    add r2,r2,r1
    /* x000 zaehlen */
    mov r1,r0
    and r1,r1,#8
    lsr r1,r1,#3
    add r2,r2,r1
    /* Anzahl in Parity-Bit umrechnen */
    and r2,r2,#1
    /* Parity-Bit anhaengen */
    lsl r0,r0,#1
    orr  r0,r0,r2
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

Aufgabe 3: Inline Assembler Code

Es ist möglich und manchmal auch sehr sinnvoll, Assemblercode in Hochsprachen wie z. B. C zu verwenden. Folgendes Beispiel zeigt Ihnen, wie man Assemblercode einbetten kann. Formulieren Sie das Programm aus Aufgabe 2 als Unterprogramm in C. Benutzen Sie dabei die Möglichkeit des Inline Assembler Code.

```
#include <stdio.h>

int mad(int i, int j, int k)
{
  int res = 0;
  __asm ("MUL␣r3,␣%[input_i],␣%[input_j]\n"
         "ADD␣%[result],␣r3,␣%[input_k]"
    : [result] "=r" (res) /* outputs: '+' = rw, '=' = wo */
    : [input_i] "r" (i), [input_j] "r" (j), [input_k] "r" (k) /* inputs */
    : "r3" /* other used registers */
  );
  return res;
}

int main(void)
{
  int a = 23;
  int b = 42;
  int c = 666;
  int d = 0;

  d = mad(a,b,c);

  printf("Result␣of␣%d␣*␣%d␣+␣%d␣=␣%d\n", a, b, c, d);
  return 0;
}
```

Lösungsvorschlag:

```
#include <stdio.h>

int append_parity(int number) {
    int res = 0;
    __asm("mov␣R2,␣#0\n"
          "mov␣%[result],␣%[input]\n"
          "and␣%[result],␣%[result],␣#1\n"
          "add␣R2,␣R2,␣%[result]\n"
          "mov␣%[result],␣%[input]\n"
          "and␣%[result],␣%[result],␣#2\n"
          "lsr␣%[result],␣%[result],␣#1\n"
          "add␣R2,␣R2,␣%[result]\n"
          "mov␣%[result],␣%[input]\n"
          "and␣%[result],␣%[result],␣#4\n"
          "lsr␣%[result],␣%[result],␣#2\n"
          "add␣R2,␣R2,␣%[result]\n"
          "mov␣%[result],␣%[input]\n"
          "and␣%[result],␣%[result],␣#8\n"
          "lsr␣%[result],␣%[result],␣#3\n"
          "add␣R2,␣R2,␣%[result]\n"
          "and␣R2,␣R2,␣#1\n"
          "mov␣%[result],␣%[input]\n"
          "lsl␣%[result],␣%[result],␣#1\n"
          "orr␣%[result],␣%[result],␣R2"
```

```
        : [result] "+r" (res) /* '+' markiert des Register als rw, '=' markiert es als wo */
        : [input] "r" (number)
        : "r2"); /* Da r2 benutzt wird, muss das hier angegeben werden (damit u.a. der Compiler weiss,
                    wie die Input- und Outputregister verteilt sind) */

    return res;
}

int main(void) {
    int number = 13;
    int number_ext = append_parity(number);
    printf("Input number: %d, Output: %d\n", number, number_ext);
}
```

Aufgabe 4: Konditionale Befehlsausführung

In der Vorlesung haben Sie folgende Möglichkeit der Abbildung eines if/then Konstrukts in Assembler kennengelernt:

```
/* Hochsprache */
if (apples == oranges)
  f = i + 1;
else
f = f - i;
----------------------
/* ARM - Assemblersprache */
/* r0=apples; r1=oranges, r2=f, r3=i */
cmp r0,r1
bne L1
add r2,r3,#1
b L2
L1:
sub r2,r2,r3
L2:
```

Der ARM-Prozessor erlaubt, wie viele moderne Prozessoren, eine sogenannte konditionale Befehlsausführung. Das bedeutet, dass Befehle nur dann ausgeführt werden, wenn bestimmte Bedingungen erfüllt sind.

a) Implementieren Sie obiges if/then Konstrukt in ARM-Assembler ohne die Nutzung von Sprungbefehlen. Zur Implementierung dieses Programms ist es sinnvoll, sich im Reference Guide[1] die Konditionen für die Befehlsausführung anzuschauen.

Lösungsvorschlag:

```
/* -- cond.s */
/* Kommentar  */
.global main /* Einsprungpunkt Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0,#13 /* apples */
    mov r1,#0  /* oranges */
    mov r2,#5  /* f */
    mov r3,#3  /* i */
    cmp r0,r1
    addeq r2,r3,#1
    subne r2,r2,r3
    mov r0,r2
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

b) Welche Vorteile hat die Übersetzung mit konditionalen Befehlen?

Lösungsvorschlag:

Der Code wird kürzer und die Ausführungsreihenfolge der Befehle ändert sich nicht.

[1] https://moodle.tu-darmstadt.de/pluginfile.php/1809290/mod_folder/content/0/Learning%20Nugget%2002%20-%20ARM%20Assembler%20Befehle/arm_instruction_set_reference_guide.pdf

Aufgabe 5: Nutzung der Statusflags

Für die Abbildung von Kontrolloperationen in Assembler oder zur Implementierung von Mehrwort-Addition stehen verschiedene Statusflags zur Verfügung. Diese sind:

- C – Carryflag (Übertragsflag)
- Z – Zeroflag (Nullflag)
- N – Negativflag (Vorzeichenflag)
- V – Overflowflag (Überlaufflag)

Der Befehl, der bei der Abbildung von Kontrolloperationen zur Setzung der Statusflags verwendet wird, heißt `cmp`. ARM-Prozessoren haben die Besonderheit, dass z. B. die Befehle zur Addition (`add`) und Subtraktion (`sub`) die Statusflags nicht automatisch setzen. Um zu erreichen, dass diese Befehle die Statusflags setzen muss dem Befehl ein s angefügt werden. Die Syntax eines Additionsbefehls lautet dann `adds`.

In der Vorlesung haben Sie folgende Möglichkeit der Abbildung eines if/then Konstrukts in Assembler kennengelernt:

```
/* Hochsprache */
if (apples == oranges)
  f = i + 1;
else
f = f - i;
----------------------
/* ARM - Assemblersprache */
/* r0=apples; r1=oranges, r2=f, r3=i */
cmp r0,r1
bne L1
add r2,r3,#1
b L2
L1:
sub r2,r2,r3
L2:
```

Implementieren Sie das Assemblerprogramm ohne die Nutzung von `cmp`.

Lösungsvorschlag:

```
/* -- flagsub.s */
/* Kommentar  */
.global main /* Einsprungpunkt Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0,#13 /* apples */
    mov r1,#0  /* oranges */
    mov r2,#5  /* f */
    mov r3,#3  /* i */
    subs r0,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 */
```

Aufgabe 6: Current Program Status Register

Das Current Program Status Register enthält u. a. auch die Statusflags. Der Befehl `mrs` kopiert den Inhalt des Current Program Status Register in ein beliebiges Register. Erweitern Sie das Programm aus Aufgabe 5 und lassen Sie sich das Zeroflag ausgeben.

Lösungsvorschlag:

```
/* -- flagsubcpsr.s */
/* Kommentar  */
.data /* Daten Bereich */
bm: .word 0x40000000 /* Bitmaske */

.global main /* Einsprungpunkt Hauptprogramm */

main:           /* Hauptprogramm */
    mov r0,#13 /* apples */
    mov r1,#0  /* oranges */
    mov r2,#5  /* f */
    mov r3,#3  /* i */
    subs r0,r0,r1
    bne L1
    add r2,r3,#1
    b L2
    L1:
    sub r2,r2,r3
    L2:
    mrs r4,cpsr /* CPSR in r4 kopieren */
    ldr r5,adr_bm /* lade Adresse von Bitmaske */
    ldr r5,[r5]  /* lade Bitmaske in r5 */
    and r4,r4,r5 /* Ausmaskierung der Bits */
    lsr r4,#30 /* 30 Bit nach rechts 1_d wenn gesetzt */
    mov r0,r4
    bx lr       /* Springe zurueck zum aufrufenden Programm */

adr_bm: .word bm /* Adresse von bm */
```