Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

5. Aufgabenblatt mit Lösungsvorschlag

22.5.2023

Konzepte der maschinennahen Programmierung, Unterprogramme, Rekursion

Aufgabe 1: Theoriefragen

(a) Erläutern Sie die Funktionsweise des Registers `sp`.

(b) Wie werden (lokale) Variablen üblicherweise realisiert?

Lösungsvorschlag:

(a) Das Register `sp` ist das sogenannte stack pointer Register. Es zeigt auf das zuletzt auf dem Stack abgelegte Element.

(b) Die Realisierung erfolgt auf dem Stack. Wenn ein Programm wenige Variablen hat, kann ein optimierenden Compiler auch Programmcode erzeugen, der auf die Nutzung des Stacks verzichtet.

Aufgabe 2: Variablentausch

Speziell bei Prozessoren und Mikrokontrollern mit einer geringen Anzahl von Registern stellt sich häufig das Problem, den Inhalt zweier Register miteinander vertauschen zu müssen, ohne dass ein weiteres, unbenutztes Register zur Verfügung steht. Eine erste Lösung des Problems liegt in der Verwendung eines Zwischenspeicherbereiches im Hauptspeicher. Eine andere Möglichkeit, die ausschließlich unter Verwendung z. B. der Register `r1` und `r2` funktioniert, besteht in der Verwendung einer logischen Verknüpfung mittels XOR.

1. Welchen Nachteil hat die Lösung mit Verwendung des Hauptspeichers oder des Stacks?
2. Schreiben Sie ein ARM-Assemblerprogramm, welches unter Verwendung des XOR den Variablentausch durchführt.[1] Erklärung zu XOR: a und b sind die Eingangsvariablen, c ist die Ausgangsvariable.

a	b	c
0	0	0
0	1	1
1	0	1
1	1	0

Lösungsvorschlag:

1. Diese Lösung hat den Nachteil, dass zwei Zugriffe auf den Hauptspeicher notwendig sind. Zugriffe auf den Hauptspeicher dauern länger, als Operationen auf Registern. Selbiges gilt für den Zugriff auf den Stack.
2. Assemblerprogramm

[1] Der Maschinenbefehl beim ARM-Assembler lautet `eor`.

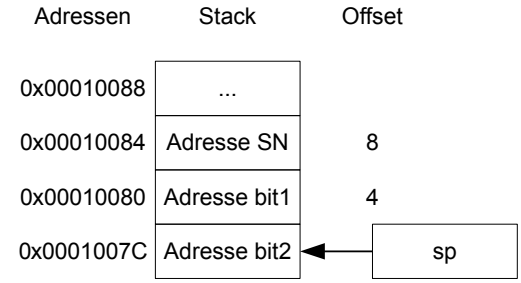
```
/* -- xchg.s */
/* Kommentar  */
.global main /* Einsprungpunkte Hauptprogramm */

main:           /* Hauptprogramm */
    mov r1, #5    /* Schreibe eine 5 in das Register r1 */
    mov r2, #12   /* Schreibe eine 12 in das Register r2 */
    eor r1,r1,r2  /* Tausche Registerinhalte */
    eor r2,r2,r1
    eor r1,r1,r2
    mov r0,r1
    bx lr       /* Springe zurueck zum aufrufenden Programm */
```

Aufgabe 3: Bitmanipulation als Unterprogramm

Implementieren Sie die Bitmanipulation (vgl. 4. Übung) als Unterprogramm. Bisher haben Sie bei Übergabe von Parametern (Zahlen-)Werte übergeben. Dies wird auch als Call-by-Value Übergabe bezeichnet. Bei der sogenannten Call-by-Reference Übergabe werden die (Speicher-)Adressen der (Zahlen-)Werte übergeben. Übergeben Sie dem Unterprogramm zur Bitmanipulation die Adressen der Seriennummer und der Bitmasken über den Stack. Es ist hilfreich, sich den Stack aufzuzeichnen und die Offsets anzugeben.

Lösungsvorschlag:




```
/* -- sn_up.s */
/* Kommentar  */
.data /* Daten Bereich */
sn: .word  0x42C27F91
bit1: .word 0x00000380
bit2: .word 0x03800000

.text

sn_test:
    /* Calling Conventions werden hier nicht umgesetzt */
    ldr r1, [sp,#8] /* lade Adresse von Seriennummer in r1 */
    ldr r1, [r1] /* lade Seriennummer in Register r1 */
    ldr r2, [sp,#4] /* lade Adresse Bitmaske1 in r2 */
    ldr r2, [r2] /* lade Bitmaske in Register r2 */
    and r1,r1,r2 /* Maskiere die Bits aus */
    lsr r1,r1,#7 /* erstes Teilergebnis */
    ldr r3, [sp,#8] /* lade Adresse von Seriennummer in r3 */
    ldr r3, [r3] /* lade Seriennummer in Register r3 */
    ldr r2, [sp] /* lade Adresse Bitmaske2 in r2 */
    ldr r2, [r2] /* lade Bitmaske in Register r2 */
    and r3,r3,r2 /* Maskiere die Bits aus */
    lsr r3,r3,#23 /* zweites Teilergebnis */
```

```
    add r4,r1,r3 /* Addition der Teilergebnisse */

    mov r0,#0
    cmp r4,#12 /* pruefe, ob Ergebnis 12 */
    bne ende /* not equal */
    mov r0,#1 /* gebe die 1 zurueck, wenn SN gueltig */
ende:
    bx lr /* zurueck zum Aufrufer */


.global main /* Definition Einsprungpunkt Hauptprogramm */

main:           /* Hauptprogramm */
    push {lr}   /* Sicherung Rueckkehradresse */
    ldr r0, adr_sn /* lade Adresse von SN */
    push {r0} /* Ablage der Adresse auf dem Stack */
    ldr r0, adr_bit1 /* lade Adresse von Bitmaske1 */
    push {r0} /* Ablage der Adresse auf dem Stack */
    ldr r0, adr_bit2 /* lade Adresse von Bitmaske2 */
    push {r0} /* Ablage der Adresse auf dem Stack */

    bl sn_test /* Sprung zu Unterprogramm */
    add sp,sp,#12 /* Stack aufraeumen */

    pop {lr}    /* Wiederherstellung Rueckkehradresse */
    bx lr       /* zurueck zum Aufrufer */

adr_sn: .word sn /* Adresse von sn */
adr_bit1: .word bit1 /* Adresse von bit1 */
adr_bit2: .word bit2 /* Adresse von bit2 */
```

Aufgabe 4: Berechnung eines Binomialkoeffizienten durch Rekursion

Einem Binomialkoeffizienten begegnet man nicht nur in der Mathematik. Dieser ist wie folgt definiert für $n, k \in \mathbb{N}$ mit $n \geq k \geq 0$

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Durch die Fakultät wird ggf. mit sehr großen Zahlen gerechnet, was schnell zu einem Overflow oder aufwändigen Rechnungen führt. Es gibt eine rekursive Definition, die mit weniger großen Zahlen auskommt. Dabei gilt

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \binom{n}{0} = \binom{n}{n} = 1$$

Implementieren Sie den rekursiven Algorithmus in C und anschließend in ARM-Assembler.

Lösungsvorschlag:

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

int BinomRec(int n, int k){
    if (n == k || k == 0)
        return 1;
    int n1 = BinomRec(n - 1, k);
    int n2 = BinomRec(n - 1, k - 1);
    return n1 + n2;
}

int main(int argc, char** argv){
    int n = 5;
    int k = 3;
    int z = BinomRec(n,k);
    printf("z=%d", z);
    return 0;
}
```

```
.text

BinomRec:
    sub sp, sp, #16     /*Speicherplatz auf dem Stack anfordern, fuer r0,r1,r2 und lr*/
    str r0, [sp, #12]   /*Sichern von r0 auf dem Stack */
    str r1, [sp, #8]    /*Sichern von r1 auf dem Stack */
    str r2, [sp, #4]    /*Sichern von r2 auf dem Stack */
    str lr, [sp]        /*Sichern von lr auf dem Stack */
    cmp r0, r1          /*1. Rekursionsanker: n == k */
    beq BinomRec1       /*Sprung zum Rekursionsende */
    cmp r1, #0          /*2. Rekursionsanker: n == 0 */
    beq BinomRec1
    sub r0, r0, #1      /*Berechnen von n-1 als Parameter fuer den 1. rekursiven Aufruf */
    bl BinomRec         /*r0 = n-1, r1 = k; rekursiver Aufruf der Funktion */
    mov r2, r0          /*Ergebnis von Berechnung in r2 speichern */
    ldr r0, [sp, #12]   /*Den Wert von n wieder vom Stack laden */
    sub r0, r0, #1      /*Berechnen von n-1 als Parameter fuer den 2. rekursiven Aufruf */
    sub r1, r1, #1      /*Berechnen von k-1 als Parameter fuer den 2. rekursiven Aufruf */
    bl BinomRec         /*r0 = n-1, r1 = k-1; rekursiver Aufruf der Funkion */
    add r0, r0, r2      /*Die Ergebnisse der beiden rekursiven Aufrufe addieren und in r0 speichern */
fin:
    ldr lr, [sp]        /*Wert von lr vom Stack laden */
    ldr r2, [sp, #4]    /*Wert von r2 vom Stack laden */
```

```
    ldr r1, [sp, #8]    /*Wert von r1 vom Stack laden, r0 nicht laden, da in r0 der Return Wert steht */
    add sp, sp, #16     /*Stackspeicher wieder freigeben */
    bx lr               /*Sprung zum Aufrufer */
BinomRec1:
    mov r0, #1          /*Rekursionsanker, es wird 0 in r0 geladen, um 0 zurueckzugeben */
    b fin               /*Zum Ende der Funktion springen */

.global main

main:
    push {lr}
    mov r0, #5          /*n = 5*/
    mov r1, #3          /*k = 3*/
    bl BinomRec         /*Funktionsaufruf von BinomRec*/
    pop {lr}
    bx lr
```