

# Fortgeschrittene Themen der IT-Sicherheit

## Aufgaben zu Einführung in die ARM Assembler Sprache

### 1 Aufgabe 1 - Verständniss

#### 1.1 Allgemeines

- Welches Speichermodell verwenden die ARM Architekturen ?
  - Load-Store Speichermodell.
- Welche Wortbreite besitzt eine Instruktion ?
  - 32 Bit unter ARM, 16 Bit unter Thumb.
- Welche der ARM Instruktionen verwenden Erweiterungsworte ?
  - Es gibt **keine** Erweiterungsworte unter ARM.
- Wie werden Instruktionen verarbeitet ?
  - Mittels Pipelining.
- Angenommen zum Zeitpunkt n wird eine ADD Instruktion, an der Adresse 0xAFF6, tatsächlich ausgeführt (execute Stufe), an welcher Adresse befindet sich der PC, wenn sich der Prozessor im ARM Modus befindet?
  - PC = 0xAFFE
- Was ist die Tool Chain? Geben pro Programmierwerkzeug eine kurze Beschreibung.
  - Assembler** Erzeugt aus einer Assembler-Source-File eine Objektdatei (relocation file).
  - Linker** Verlinkt (mehrere) Objektdateien zu einem ausführbaren Programm.
  - Debugger** Programm zur Analyse und Kontrolle einer ausführbaren Datei.
- Nennen Sie drei mögliche Sektionen, in die eine ausführbare Datei untergliedert ist und beschreiben Sie diese kurz.
  - text** Enthält den auszuführenden Programmcode
  - data** Enthält initialisierte, schreibbare Daten
  - bss** Enthält uninitialisierte, schreibbare Daten
  - rodata** Read only Daten Sektion
- Zeichnen Sie das typische Speicher-Layout eines Prozesses unter Linux.

## 1.2 Bits and Bytes

- Welche Größe (in Bit) besitzen: `BYTE`, `HALF WORD`, `WORD`, `DOUBLE WORD`/ `Giant`?
  - 1 Byte, 2 Byte, 4 Byte, 8 Byte
- Welche Endianness wird unter ARM genutzt?
  - ARM ist Bi-Endian
    - \* little-endian
    - \* big-endian
- Beschreiben Sie `Little Endian` in einem Satz.
  - Das LSB steht an der kleinsten Speicheradresse eines Wortes

## 1.3 Register

- Was sind `Callee Save Register`?
  - Register, deren Werte über einen Unterprogrammaufruf hinweg, von den aufgerufenen Subroutine, erhalten werden müssen.
- Welches Register (Rx) stellt den Program Counter dar?
  - `R15/ pc`
- Was ist der Zweck des Link Registers? Was ist in dieser Hinsicht der Unterschied zu x86?
  - Bei einem Unterprogrammaufruf mittels `BL/ BLX` wird die Rücksprungadresse in das Link Register (LR) geschrieben. Das Register kann dann entweder direkt zum Zurückkehren aus einer Subroutine verwendet werden( `BLX lr` [ Nur für Leaf Funktionen ] ) alternativ kann die Rücksprungadresse auch z.B. auf den Stack gepushed werden.
  - Unter X86 wird bei einem Unterprogrammaufruf (`call <label>`) die Rücksprungadresse direkt auf den Stack gepushed.

## 2 Data Processing Instructions

Für die nachfolgenden Aufgaben wird ein Arm Computer, wie z.B. Raspberry Pi, oder ein Emulator, wie z.B. Qemu, benötigt, sowie ein Linux Betriebssystem.

Die Beschreibungen beziehen sich auf eine ARM Entwicklungsumgebung bestehend aus eine `Raspberry Pi 3` auf dem `Raspbian` installiert wurde.

## 2.1 Einrichten der Entwicklungsumgebung

Sie können den Raspberry Pi mittels HDMI und USB direkt mit einem Monitor, sowie Tastatur verbinden.

Als Alternative können Sie sich auch mittels **SSH**, von einem anderen Computer aus, mit dem Raspberry Pi verbinden. Dafür müssen Sie sich im selben Netzwerk befinden. Für den Verbindungsaufbau benötigen Sie weiterhin die IP-Adresse des Gerätes. Angenommen Sie möchten sich als Nutzer **pi** auf dem Gerät mit der IP-Adresse **192.168.62.255** einloggen, dann können Sie dies über die Konsole mit folgendem Befehl bewerkstelligen:

```
ssh pi@192.168.62.255
```

**ACHTUNG:** Neuere Versionen von Raspbian haben **ssh standardmäßig ausgeschaltet**. Sie müssen dies in den Einstellungen ggf. zuerst ändern um von einem anderen Gerät per SSH auf den Raspberry Pi zugreifen zu können.

Öffnen Sie auf dem Raspberry Pi eine Konsole (strg + alt + t) und starten sie den SSH Service:

```
sudo systemctl enable ssh
sudo systemctl start ssh
```

Weitere Möglichkeiten

## 2.2 Bearbeitung der Aufgaben

Für die Bearbeitung der Aufgaben steht Ihnen eine Template Datei namens **template.s** zur Verfügung. Diese kann verwendet werden um die einzelnen Aufgaben zu bearbeiten.

In der **main** Subroutine befindet sich bereits Beispielcode, der ersetzt werden kann. Außerdem können natürlich weitere Funktionen definiert werden.

Sie können Ihren Source Code mithilfe des Compilerscripts **compile.sh** compilieren, dabei wird der Code von **debug.c** mit verlinkt, der Ihnen die Möglichkeit bietet, Ihre Resultate auf der Kommandozeile auszugeben.

```
./compile.sh template.s
```

Um Ihr Ergebnis auszugeben, können Sie die **info()** funktion aufrufen. Diese erwartet als erstes Argument das Ausgabe Format, d.h. Hexadezimal, Oktal, vorzeichenbehafteter Integer, vorzeichenloser Integer und als zweites Argument den auszugebenden Wert.

```
void info(const int func, const int val);
```

Angenommen es wurde eine simple Additionsfunktion implementiert und das Ergebnis soll nun als Hexadezimalwert ausgegeben werden, dann kann **info** wie folgt aufgerufen werden.

Vorbedingung: Der auszugebende Wert befindet sich in R0

```
mov r1, r0    @ pass value as second argument
mov r0, #HEX  @ specify output format
bl  info      @ call info
```

## 2.3 Daten und Funktionen

### 2.3.1 a)

Auf modernen Systemen ist ein Dividierwerk standard, mit dem eine Zahl durch eine andere, mittels Hardware, geteilt werden kann. Die Armv-8 Architektur bietet die Instruktionen UDIV und SDIV um vorzeichenlose bzw. vorzeichenbehaftete Zahlen zu dividieren. Ältere Architekturen besaßen diese Möglichkeit nicht. Dort werden Divisionen in Software, d.h. durch eine Funktion, realisiert. Solche Dividieralgorithmen sind jedoch sehr ineffizient und benötigen viele Taktzyklen.

Berechnen Sie  $(8 * 5) / 2$  mit so wenigen Instruktionen wie möglich, ohne die Verwendung eines Multiplikations- oder Dividierbefehls und ohne die einzelnen Zahlen im Voraus zusammenzufassen.

Folgende Instruktionen sind vorgegeben:

```
mov r0, 0x8
```

**Lösung:**

```
mov r0, 0x8
add r0, r0, lsl #0x2
lsr r0, #0x1
```

### 2.3.2 b)

Für einen Zyklischen Buffer der Größe `buffer_size` der mittels `offset` indiziert wird, könnte eine Erhöhung um `n` Wörter wie folgt aussehen:

```
offset = (offset + n) % buffer_size;
```

Angenommen der obige Ausdruck benötigt 50 Taktzyklen zur Berechnung (aufgrund der Modulo Rechnung), geben Sie eine, Ihrer Meinung nach, effizientere Lösung an. Implementieren Sie Ihre Lösung in Assembly. Wie viele effektive Taktzyklen benötigt Ihre Implementierung unter der Annahme, dass jede Operation auf Daten (ausgenommen Division, Multiplikation, Load und Store) einen effektiven Taktzyklus benötigt?

**Lösung:**

Effizientere Lösung in C

```
if( ( offset = offset + n ) >= buffer_size )
{
    offset = offset - buffer_size
}
```

Implementierung in Assembly

```
offset      .req r0
n           .req r1
buffer_size .req r2

add offset, n           @ 1 Takt
cmp offset, buffer_size @ 1 Takt
subge offset, buffer_size @ 1 Takt
```

### 2.3.3 c)

Nun soll eine einfache Funktion, in Assembler, implementiert werden. Diese soll einen Integer als Argument übergeben bekommen und diesen dann, ohne Beachtung des Restes, durch zwei teilen und das Ergebnis zurückgeben. Achten Sie auf eine möglichst effiziente Implementierung.

```
int
div_two(int x)
{
    return x / 2;
}
```

Rufen Sie Ihre Funktion mit unterschiedlichen Werten auf und überprüfen Sie die Ergebnisse. Sie können dafür die externe **info** Funktion verwenden.

**Lösung:**

```
div_two:
    asr r0, #0x1
    blx lr
```

## 2.4 Load and Store

Nun wollen wir uns explizit mit dem Speichern und Laden von Daten auseinandersetzen.

### 2.4.1 d)

Definieren Sie eine Integer-Variable mit einem beliebigen Wert in der **.data** section. Schreiben Sie ein kleines Programm, das diesen Wert inkrementiert.

Nutzen Sie bei Bedarf die externe **info** Funktion.

**Lösung:**

```
.extern info

.equ HEX, 0
.equ OCT, 1
.equ UNSIGNED, 2
.equ SIGNED, 3

.data
@ Your initialized data
i:    .4byte    0xf

.bss
@ Your uninitialized data

.text

.global main
main:
```

```

@ load addr of i using a literal pool
ldr r4, literal_pool

@ load the value of i
ldr r5, [r4]

@ increment the value of i
add r5, r5, #1

@ update i
str r5, [r4]

@ exit
eor r0, r0 @ r0 := 0
mov r7, #0x1 @ r7 := SYS_exit
svc 0 @ exit 0

literal_pool:
.word i

```

Beachten Sie, dass Sie den Wert von `i` nicht direkt mittels `ldr` aus der `.data` section laden können. Um dieses Problem zu umgehen wurde ein sog. Literal Pool definiert. Innerhalb von diesem wurde die Adresse von `i` hinterlegt. Mittels `ldr` kann die Adresse von `i` nun in ein Register (`r4`) geladen werden. `r4` wird dann genutzt um auf den eigentlichen Wert von `i` zuzugreifen.

Alternativ kann auch die Pseudoinstruktion **`ldr Rd, =i`** verwendet werden. Diese legt implizit einen Literal Pool an und referenziert `i` über diesen.

`r4` bzw. `r5` wurden nicht ohne Grund gewählt. Da zwischen den Instruktionen `info()` aufgerufen wird werden zwei callee save Register genutzt um sicher zu gehen, dass die Werte erhalten bleiben.

### 2.4.2 e)

Mittels der **`ldm`** (load multiple) Instruktion können mehrere Werte gleichzeitig geladen werden. Die Laufzeit der Instruktion hängt dabei von der Anzahl der verwendeten Register ab.

Definieren Sie drei beliebige Variablen in der **`.data`** section, summieren Sie diese auf und geben Sie das Ergebnis mittels `info()` auf der Konsole aus.

#### Lösung:

```

.extern info

.equ HEX, 0
.equ OCT, 1
.equ UNSIGNED, 2
.equ SIGNED, 3

.data
@ Your initialized data
array: .word 2
       .word 3

```

```

        .word 5

.bss
@ Your uninitialized data

.text

.global main
main:
    @ load address of array
    ldr r0, literal

    @ load all three values in a row
    @ r1 := *(r0 + 0)
    @ r2 := *(r0 + 4)
    @ r3 := *(r0 + 8)
    ldmia r0, {r1, r2, r3}

    @ sum up all values
    add r1, r1, r2
    add r1, r1, r3

    @ out
    mov r0, #HEX
    bl info

    eor r0, r0 @ r0 := 0
    mov r7, #0x1 @ r7 := SYS_exit
    svc 0 @ exit 0

literal: .word array

```

Zu Anfang wird ein Array in der .data section definiert, das drei Wörter umfasst. Die Adresse des Arrays wird mittels Literal Pool geladen. Danach werden mittels ldmia (load multiple increase after) alle drei Werte gleichzeitig in die Register r1 bis r3 geladen. r0 wird dabei als Basisadresse genutzt. Schlussendlich werden die geladenen Werte aufsummiert und auf der Kommandozeile in hexadezimal ausgegeben.

### 2.4.3 f)

Der **Bubble Sort** Algorithmus kann zum sortieren von Listen/ Arrays (z.B. Zahlen) verwendet werden und kann u.a. wie folgt implementiert werden:

```

bubbleSort(list):
    n := length(list)
    while n > 1:
        for i := 0 to n-1 do:
            if list[i] > list[i+1]:
                swap(list[i], list[i+1])
            end if
        end for
        n := n - 1
    end while

```

```
end bubbleSort
```

Die Datei **bubble.s** enthält die Definition eines Array von Zahlen. Implementieren Sie eine Form des Bubble Sort Algorithmus und sortieren Sie das Array aufsteigend.