Name der Vorlesung
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 Enthällt den auszuführenden Programmcode

  data Enthällt initialisierte, schreibbare Daten

  bss Enthällt 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-Endidan
    - 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. Quemu, 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` einlogen, dann können Sie dies über die Konsole mit folgendem Befehl bewerkstelligen:

```
ssh pi@192.168.62.255
```

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 `compiel.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 Dividieren mal anders

2.3.1 a)

Auf modernen Systemen ist ein Dividierwerk standard, mit dem ein 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 besitzten 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 einzelnene Zahlen im Vorraus 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
```