

Rechnerorganisation

Zusammenfassung SS21

Felix Marx

22. März 2021

Inhaltsverzeichnis

1	Vorlesung 1	3
1.1	Begriffe	3
1.2	Geschichte	4
1.3	Ethik	4
2	Vorlesung 2	4
2.1	Speicher	4
2.2	Einführung Assembler	5
2.2.1	Übersetzung und Ausführung	6
3	Vorlesung 3	8
3.1	Grundlegende Assembler Befehle	8
3.1.1	load	8
3.1.2	store	8
3.1.3	mov	8
3.1.4	add	8
3.1.5	sub	9
3.1.6	mul	9
3.1.7	lsl	9
3.1.8	lsr	9
3.2	Flags	9
3.3	Assembler Sprungbefehle	10
3.3.1	Unbedingte Sprünge	10
3.3.2	beq	10
3.4	If-Verzweigungen	10
3.5	If-Else-Verzweigungen	11
3.6	while-Schleifen	11
3.7	for Schleifen	11
4	Vorlesung 4	12
4.1	Nutzen des Hauptspeichers	12
4.2	Arrays	12
4.3	Unterprogramme	13
5	Vorlesung 5	13
5.1	Stack	13

5.2	Rekursion	14
6	Vorlesung 6	15
6.1	OpenMP	15
6.2	Assembler	15
6.3	Objekt-Programme	16
6.4	Binder und Lader	17
6.5	Laufzeit	17

1 Vorlesung 1

1.1 Begriffe

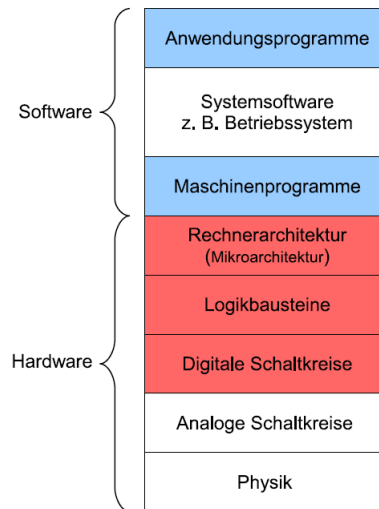


Abbildung 1: Das Schichtenmodell ist eine abstrakte Darstellung eines Computers

- Abstraktion versteckt unnötige Details
- Schichtenmodell teilt Dienstleistungen Schichten zu
 - Schichten verrichten Arbeit für die nächst höhere
 - Vorteile:
 - * Austauschbarkeit von Schichten
 - * Erfordert i.d.R. nur Kenntnis der aktuellen Schicht
 - * z.B. Gerätetreiber erfordern Wissen niedrigerer Schichten
 - Nachteile:
 - * ggf. geringere Leistungsfähigkeit

Computer ist ein Synonym für Datenverarbeitungssystem bzw. Rechnersystem:

*”Ein Datenverarbeitungssystem ist eine Funktionseinheit zur Verarbeitung und Aufbewahrung von Daten. Verarbeitung umfasst die Durchführung **mathematischer, umformender, übertragender und speichernder** Operationen.”*

Ein Rechnersystem unterscheidet sich von Messgeräten, dadurch dass es über ein ladbares Programm schrittweise eine Funktion ausführt.

Minimale Komponenten eines Rechnersystems:

- Prozessor
- Speicher
- I/O

1.2 Geschichte

Bezeichnung	Technik und Anwendung	Zeit
Abakus, Zahlenstäbchen	mechanische Hilfsmittel zum Rechnen	bis ca. 18. Jahrhundert
mechanische Rechenmaschinen	Apparate zum Rechnen	1623 bis ca. 1960
elektronische Rechanlagen	Lösen numerischer Probleme	seit 1944
Datenverarbeitungsanlage	Texte und Bilder bearbeiten	seit ca. 1955
Informationsverarbeitungssystem	Bilder und Sprache erkennen (KI)	seit 1968

Zuse Rechner:

- Z1 (1937): mechanische Rechenmaschine, bestehend aus
 - Ein-/Ausgabewerk
 - Rechenwerk
 - Speicherwerk
 - Programmwerk, Programm auf gelochten Filmstreifen
- Z2 (1939): Austausch der mechanischen Schaltglieder durch Relais (ca. 200) 10 Hz
- Z3 (1941): erster funktionsfähiger Digitalrechner weltweit

1.3 Ethik

- Dual Use Problem (Ziviler und militärischer Einsatz)
- Digitale Souveränität bezeichnet die Fähigkeit zu selbstbestimmtem Handeln und Entscheiden im digitalen Raum
 - Selbstbestimmtes Handeln und Entscheiden
 - Digitale Medien souverän zu nutzen (Medienkompetenz)
 - Fähigkeit Vertrauenswürdigkeit und Integrität der Datenübertragung, -speicherung, -verarbeitung durchgängig zu kontrollieren
 - Selbstbestimmung über die Nutzungsbestimmungen für seine Daten
 - eigene Fähigkeiten mit Schlüsseltechnologien eigens Dienste und Plattformen zu betreiben

2 Vorlesung 2

2.1 Speicher

Speicher können danach klassifiziert werden, ob sie für den Programmierer **explizit** oder nur **implizit/transparent** zugreifbar sind.

- explizite Nutzung
 - interner Prozessorspeicher
 - * schnell, zur temporären Speicherung von Maschinenbefehlen und Daten
 - * direkter Zugriff durch Maschinenbefehle

- * Halbleiter ICs¹
- Hauptspeicher
 - * relativ groß und schnell, für Programme und Daten während der Ausführung
 - * direkter Zugriff durch Maschinenbefehle
 - * Halbleiter ICs
- Sekundärspeicher
 - * sehr groß und langsam, für permanente Speicherung von Programmen und Daten
 - * indirekter Zugriff über I/O-Programme zur Übertragung in den Hauptspeicher
 - * Halbleiter ICs, Magnetplatten, optische Laufwerke, Magnetbänder
- transparente Nutzung
 - bestimmte Register auf dem Prozessor
 - Cache Speicher

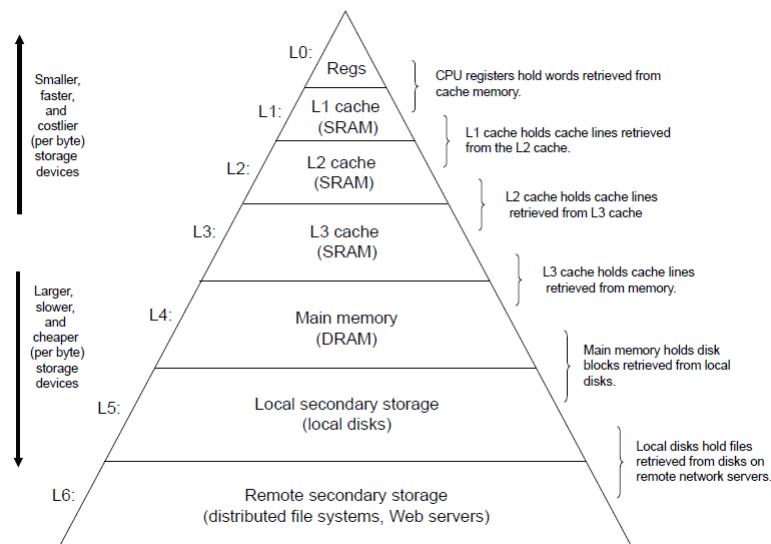


Abbildung 2: Speicherhierarchie, die Größe der Speicher nimmt nach oben ab, die Geschwindigkeit nimmt zu

2.2 Einführung Assembler

Maschinensprache (Assembler) ist ein primitives Paradigma².

Das Programmiermodell von Assembler bezeichnet den Registersatz eines Prozessors, bzw. die Register, die durch Programme angesprochen werden können sowie die verfügbaren Befehle (Befehlssatz). Der Instruction Pointer (IP) und Program Counter (PC) zählen nicht zum Registersatz des Prozessors.

Komponenten des Rechnersystems:

¹IC - Integrated circuit

²Paradigma bezeichnet ein übergeordnetes Prinzip, welches sich in Beispielen manifestiert

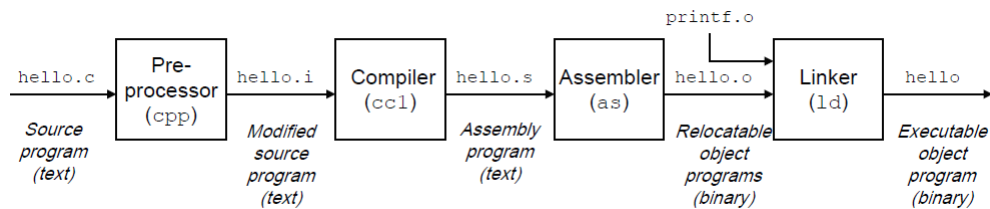


Abbildung 3: Die Phasen der Übersetzung eines C Programmes in binären Maschinencode

- CPU/Prozessor: führt im Hauptspeicher abgelegte Befehle aus
- ALU: Ausführung der Operationen
- PC: Programmzähler, der auf den nächsten Maschinenbefehl im Hauptspeicher zeigt
- Register: Schneller Speicher für Operanden
- Hauptspeicher: Speichert Befehle und Daten
- Bus Interface: Verbinden der einzelnen Komponenten

2.2.1 Übersetzung und Ausführung

1. Preprocessor

- Aufbereitung durch Ausführung von Direktiven (mit #) Inhalt der referenzierten Datei wird in Programmdatei kopiert
- Ausgabe: C-Programm mit der Endung .i

2. Compiler

- Übersetzt das C-Programm name.i in ein Assemblerprogramm name.s

3. Assembler

- Übersetzt name.s in Maschinsprache in das Objekt-Programm hello.o

4. Linker

- Zusammenführen verschiedener Module (z.B. printf Funktion)
- Module werden zu einem ausführbaren Programm kombiniert
- Ausgabe des Bindeprogramms: name Datei, welche eine ausführbare Objekt-Datei, die in den Speicher geladen und ausgeführt werden kann

Nach dem Übersetzen kann die Objekt-Datei über die Shell ausgeführt werden. Dabei werden zunächst die Zeichen des Kommandos in die Register gelesen und den Inhalt dann in den Hauptspeicher gespeichert.

Anschließend werden Befehle und Daten schrittweise von der Festplatte in den Hauptspeicher kopiert.

gcc programm.c übersetzt das C-Programm

gcc -S programm.c generiert das Assemblerprogramm

Verschiedene Optimierungseinstellungen des Compilers lassen sich mit -O1 und -O2 dazuschalten.

Konstante Ausdrücke werden ggf. zur Compilezeit bereits ausgewertet und in Assembler durch das Ergebnis ersetzt.

Wir unterscheiden die Befehle eines Rechnersystems in **CISC** - Complex Instruction Set Computer und **RISC** - Reduced Instruction Set Computer. CISC besitzen viele komplexe Befehle, wohingegen RISC weitgehend Befehle mit identischer Ausführungszeit hat um effizientes Pipelining zu ermöglichen. RISC werden auch als Load/Store-Architekturen bezeichnet.

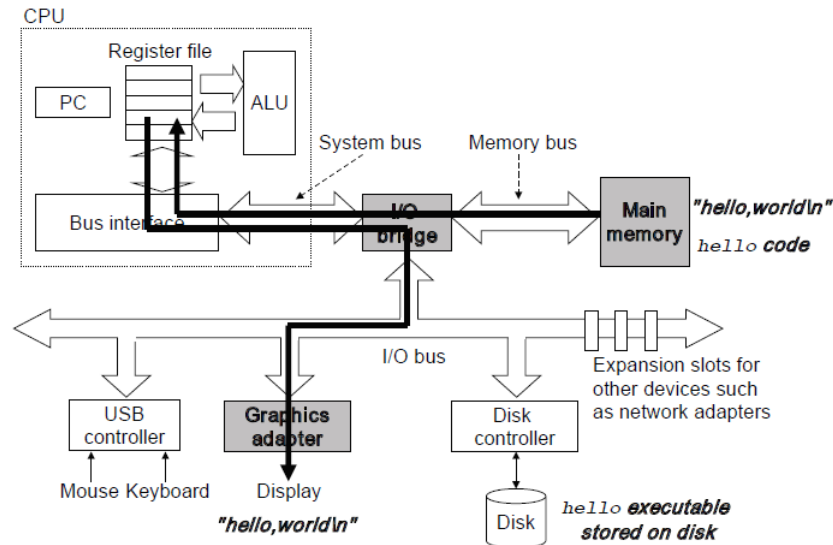


Abbildung 4: Im letzten Schritt der Ausführung werden die Maschinenbefehle des Programmes ausgeführt

Weiterhin beeinflusst die Struktur eines Prozessors die Leistungsfähigkeit und Kosten des Rechnersystems massiv.

Wir teilen Rechnersysteme nach der Anzahl der Operanden in einem Maschinenbefehl ein (n-Adressmaschinen).

CISC-Maschine: Intel Architektur
 2-Adressmaschine: Intel Architektur
 RISC-Maschine: ARM Architektur
 3-Adressmaschine: ARM Architektur

ARM besitzt 16 Register und ein Current Processor Status Register (CPSR). Dabei sind:

- R0: das Rückgaberegister
- R1 - R12: frei verwendbar
- R13: der Stack Pointer (sp) zeigt auf den Kopf des Stacks
- R14: das Link Register (lr) zeigt auf die Rücksprungadresse
- R15: der Programm Counter (pc) speichert die aktuelle Programmadresse

Wir unterscheiden in der Speicherorganisation: Big-Endian und Little-Endian.

Bei **Big-Endian** werden die Bytes vom höchstwertigen Ende an gezählt.

Bei **Little-Endian** werden die Bytes vom niedrigstwertigen Ende an gezählt.

In der Adressnummerierung hat bei Big-Endian das MSB also die Adresse 0, bei Little-Endian hat das LSB die Adresse 0.

ARM ist byte-adressiert und verwendet i.d.R. Little-Endian (kann aber manuell festgelegt werden). Ein Wort ist dabei 32 Bit = 4 Bytes lang, somit sind Wortadressen immer Vielfache von 4.

3 Vorlesung 3

3.1 Grundlegende Assembler Befehle

Direktwerte, d.h. Werte welche direkt als solche im Assemblercode geschrieben werden, müssen im Zweierkomplement als 32 Bit Wert repräsentierbar sein.

3.1.1 load

Mit dem Ladebefehl `ldr` (load word) wird ein Datenwort von einer Speicheradresse gelesen und in das angegebene Register geschrieben.

```
1 ldr r1, [r2, #4]
```

Im obigen Beispiel wird ein Datenwort von der Adresse ($r2 + 4$) gelesen und in das Register `r1` geschrieben. Also das zweite³ Datenwort im Register `r2`.

Gelesen als Basisadresse `r2` plus Offset 4.

Als Basisadresse darf jedes Register verwendet werden.

Der Offset kann auch Hexadezimal angegeben werden (z.B. `#0xC` für `#12`).

Auch eine Angabe durch ein Register ist erlaubt.

3.1.2 store

Mit dem Speicherbefehl `str` (store word) wird ein Datenwort in das zweite Register plus Offset geschrieben.

```
1 str r2, [r3, #0x14]
```

Im obigen Beispiel wird das an der Adresse von `r2` gespeicherte Datenwort an die Adresse `r3` plus $20 = 0x14$ geschrieben. Der Befehl hat auch eine Byte Variante `strb`, welche das LSB der gegebenen Adresse an das Ziel schreibt.

3.1.3 mov

Schreibt den angegebenen Wert in das Zielregister.

```
1 mov r1, #0
```

Im obigen Beispiel wird der Wert 0 an die Adresse `r1` geschrieben.

3.1.4 add

Addiert zwei Werte zusammen und schreibt das Ergebnis an die Ergebnisadresse.

```
1 add r0, r0, #4
```

Im obigen Beispiel wird der Wert an der Adresse `r0` um 4 erhöht und dann an die Adresse `r0` geschrieben.

³Da die Datenworte 4 Byte lang sind

3.1.5 sub

Subtrahiert zwei Werte von einander und schreibt das Ergebnis an die Ergebnisadresse.

```
1 sub r3, r6, r9
```

Im Beispiel wird der Wert an der Adresse r9 vom Wert an der Adresse r6 subtrahiert und an die Adresse r3 geschrieben.

3.1.6 mul

Multipliziert die beiden Operanden mit einander und speichert die least significant 32 Bit an die Ergebnisadresse.

```
1 mul r1, r2, r3
```

Im Beispiel wird der Wert an der Adresse r2 mit dem Wert an der Adresse r3 multipliziert und an die Adresse r1 geschrieben.

3.1.7 lsl

Ein logischer Shift nach links⁴. Entspricht einer Multiplikation um die Zahl 2^n . Dabei wird der Wert im zweiten Register um den Wert im dritten Operanden nach links geshiftet.

```
1 lsl r1, r2, r3
2 lsl r1, r2, #1
```

Wenn der dritte Operand ein Register ist, wird nur das least significant byte beachtet. Ist der Operand ein Direktwert, so darf dieser maximal 32 sein.

3.1.8 lsr

Ein logischer Shift nach rechts. Entspricht einer Division um die Zahl 2^n . Dabei wird der Wert im zweiten Register um den Wert im dritten Operanden nach rechts geshiftet.

```
1 lsr r1, r2, r3
2 lsr r1, r2, #1
```

Die selben Einschränkungen wie bei lsl gelten.

3.2 Flags

Wichtige Flags sind:

- CF Übertragsflag (Carryflag)
- ZF Nullflag (Zeroflag)
- SF Vorzeichenflag (Signflag)
- OF Überlaufflag (Overflowflag)

Unterscheidung zwischen CF und OF Flags:

Overflowflag (OF) wird gesetzt, wenn wir z.B. zwei positive Zahlen addiert werden, aber aufgrund eines Speicherüberflusses das Ergebnis negativ ist.

⁴Beim logischen Shift wird der Überfluss abgeschnitten und die leeren Stellen mit 0'en aufgefüllt

Carryflags (CF) werden gesetzt, wenn wir z.B. eine positive und eine negative Zahl addieren, und das Ergebnis noch korrekt darstellbar ist, aber in der binären Addition ein Bit überläuft. Zum Beispiel so:

```

    0101    5
    1111   -1
  -----
1 0100    4

```

Ist das Ergebnis negativ wird das Signflag (SF) auf 1 gesetzt.

3.3 Assembler Sprungbefehle

3.3.1 Unbedingte Sprünge

Unbedingte Sprünge (b) werden immer ausgeführt sobald sie im Programmablauf erreicht werden.

Die Sprungadressen werden mit Labels im Programmcode festgelegt und mit einem Doppelpunkt abgeschlossen.

```

1 mov r1, #4
2 b label
3 sub r1, r1, #2
4 label:
5 ...

```

3.3.2 beq

Der Sprung wird nur ausgeführt falls das Zeroflag (ZF) gesetzt ist.

Dabei wird i.d.R. vor dem Sprungbefehl der Vergleichsbefehl cmp aufgerufen, welcher zwei Werte durch Subtraktion vergleicht. Der Befehl cmp muss als ersten Parameter immer ein Register bekommen, der zweite Parameter ist beliebig.

```

1 mov r0, #4
2 cmp r0, #8 /* setzte Flags auf Basis von r0 - 8 = -4 ⇒ NZCV = 1000 */
3 beq label /* Hier kein Sprung, da Z != 1 */
4 ...
5 label:

```

3.4 If-Verzweigungen

In Hochsprache:

```

1 if (r0 == r1)
2     r2 = r3 + 1;
3 r2 = r2 - r3;

```

In Assembler negieren wir die Bedingung, sodass falls diese erfüllt ist wir den konditionalen Block überspringen können.

```

1 cmp r0, r1
2 bne L1 /* Springt falls Werte ungleich ⇔ Z = 0 */
3 add r2, r3, #1
4 L1:
5 sub r2, r2, r3

```

3.5 If-Else-Verzweigungen

In Hochsprache:

```
1 if (r0 == r1)
2     r2 = r3 + 1;
3 else
4     r2 = r2 - r3;
```

In Assembler negieren wir wieder die Bedingung, um somit direkt in den else Case zu springen, hängen ans Ende des if-Blockes noch einen unbedingten Sprungbefehl, um den else-Block zu überspringen.

```
1 cmp r0, r1
2 bne L1
3 add r2, r3, #1
4 b L2
5 L1:
6 sub r2, r2, r3
7 L2:
```

3.6 while-Schleifen

In Hochsprache:

```
1 int pow = 1;
2 int x = 0;
3 while (pow != 128) {
4     pow = pow * 2;
5     x = x + 1;
6 }
```

In Assembler überprüfen wir nach unserer Einstiegsmarke, ob die negierte Fortsetzungsbedingung erfüllt ist und beenden die Ausführung im positiven Fall. Sonst wird der Schleifenkörper ausgeführt und wir springen unbedingt zur Einstiegsmarke zurück.

```
1 mov r0, #1
2 mov r1, #0
3 WHILE:
4 cmp r0, #128
5 beq DONE
6 lsl r0, r0, #1 /* pow = pow * 2 */
7 add r1, r1, #1 /* x = x + 1 */
8 b WHILE
9 DONE:
```

3.7 for Schleifen

In Hochsprache:

```
1 int sum = 0;
2 int i;
3 for (i = 0; i < 10; i = i + 1) {
4     sum = sum + i;
5 }
```

In Assembler prüfen wir direkt nach der Einstiegsmarke, ob die negierte Fortsetzungsbedingung erfüllt ist und beenden im positiven Fall. Sonst wird der Schleifenkörper ausgeführt und im Anschluss Zähler entsprechend angepasst. Dann springen wir unbedingt zur Einstiegsmarke zurück.

```

1 mov r1, #0
2 mov r0, #0
3 FOR:
4 cmp r0, #10
5 bge DONE /* Springe falls der gilt r0 > 10 ⇔ N != 0 && Z = 0 */
6 add r1, r1, r0 /* sum = sum + i */
7 add r0, r0, #1 /* i = i + 1 */
8 b FOR
9 DONE:

```

4 Vorlesung 4

4.1 Nutzen des Hauptspeichers

Anstatt Werte als Direktwerte zu benutzen, können sie auch als Variablen im Datenbereich des Programms definiert werden.

```

1 .data /* Datenbereich */
2 var1: .word 5 /* Variable 1 im Speicher mit Wert 5 */
3 var2: .word 12
4 var3: .word 15
5
6 .global main /* Definition Einsprungpunkt Hauptprogramm */
7
8 main: /* Hauptprogramm */
9     ldr r0, var1 /* laedt Wert von var1 in r0 */
10
11     ldr r1, adr_var2 /* laedt Adresse von var2 in r1 */
12     ldr r2, [r1] /* Laedt den Inhalt von Adresse r1 in r2 */
13
14     ldr r3, =var3 /* Laedt die Adresse von var3 direkt in r3 */
15     ldr r4, [r3] /* Laedt den Inhalt von Adresse r3 in r4 */
16
17     add r0, r0, r2
18     bx lr /* Springe zurueck zum aufrufenden Programm */
19
20 adr_var2: .word var2 /* Adresse von Variable 2 */

```

Die Daten in var1 werden direkt adressiert, die Daten in var2 werden indirekt adressiert. Die Abkürzung für die Adresse einer im Datenbereich definierten Variable ist: =variablenname, wie bei var3 dargestellt.

Die einzelnen Registerfelder liegen direkt übereinander, sodass wenn wir Zeile 15 durch:

```

1 ldr r4, [r3, #8]

```

ersetzen, die Adresse $(r3 + 8) = (r2 + 4) = r1$ erhalten und von dieser Adresse den Wert 12 an r4 schreiben.

4.2 Arrays

Arrays oder auch Datenfelder können als konsekutive Datenwörter aufgefasst werden welche übereinander im Speicher liegen, sodass der Index eines einzelnen Feldes dem Offset von der Basisadresse entspricht.

Dabei lädt man i.d.R. die Basisadresse des Arrays in ein Register und den Index in ein weiteres. Um nun auf einzelne Datenwörter zuzugreifen, muss der Index um die Wortbreite (4 Bytes) multipliziert werden.

```

1 mov r0, #0x11111 /* Basisadresse des Arrays */
2 mov r1, #0 /* Index i */
3 ...
4 lsl r2, r1, #2 /* r2 = i * 4 (Offset) */
5 ldr r3, [r0, r2] /* Laedt das indizierte Arrayelement */

```

4.3 Unterprogramme

Unterprogramme helfen bei der strukturierten Programmierung, indem Teile des Hauptprogrammes in Teilprogramme (TP) ausgelagert werden.

Man unterscheidet zwischen Makrotechnik und Unterprogrammtechnik. Bei der **Makrotechnik** wird das Programm an den benötigten Stellen einkopiert, dabei wird dem TP, dem sogenannten Makro, ein Name zugeordnet (Makroname). An den Stellen wo das Makro einkopiert werden soll, wird der Makroname genannt.

Bei der **Unterprogrammtechnik** ist das Unterprogramm (UP) nur einmal im Code vorhanden. Es wird durch eine Marke (Unterprogrammname) gekennzeichnet. Soll das UP aufgerufen werden, erfolgt ein Sprungbefehl mit der Marke als Operand. Am Ende des UPs erfolgt die Rückkehr in das aufrufende Programm mittels speziellem Sprungbefehl (z.B. bx). Die Rückkehradresse wird in Registern oder auf dem Stack gespeichert.

Lokale Variablen werden i.d.R. über den Stack realisiert. Als Regel für Prozeduraufrufe gilt:

- Der Aufrufer übergibt Argumente an den Aufgerufenen und springt zum Aufgerufenen
- Der Aufgerufene führt eine Funktion/Prozedur aus und gibt das Ergebnis an den Aufrufer zurück (r0)
- Die Rücksprungadresse liegt direkt hinter der Aufrufstelle
- Der Aufgerufene darf keine Register oder Speicherstellen überschreiben, die im Aufrufer genutzt werden
- Aufgerufene Unterprogramme dürfen keine unbeabsichtigten Seiteneffekte haben

```

1 main:
2 mov r0, #4 /* Argument 0 ist 4 */
3 mov r1, #30 /* Argument 1 ist 30 */
4 bl sum /* Funktionsaufruf, bl legt Rueckkehradresse in r14 (lr) */
5 mov r2, r0 /* r0 ist der Rueckgabewert */
6
7 sum: /* Unterprogrammname */
8 add r2, r0, r1 /* Fuehrt seine Funktion aus */
9 mov r0, r2 /* Legt Rueckgabewert in r0 ab */
10 mov pc, lr /* Ruecksprung zum Aufrufer */

```

5 Vorlesung 5

5.1 Stack

Der Stack wächst bei ARM nach unten, d.h. von hohen zu niedrigen Speicheradressen. Der Stackpointer: r13 (sp) zeigt auf das oberste Element des Stacks.

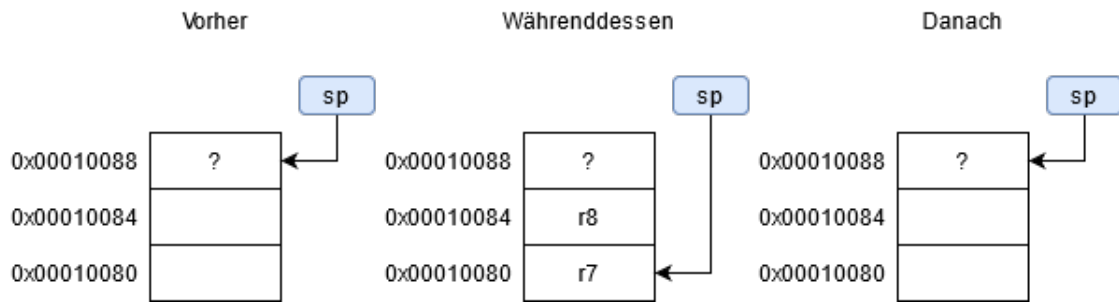


Abbildung 5: Repräsentation des Stacks vor, während und nach der Programmausführung

```

1 sub sp, sp, #8 /* reserviert Speicher fuer 2 Woerter auf dem Stack */
2 str r8, [sp, #4] /* sichert den Wert von r8 auf dem Stack */
3 str r7, [sp]
4 ...
5 ldr r7, [sp] /* stellt den Wert von r7 wieder her */
6 ldr r8, [sp, #8] /* stellt den Wert von r8 wieder her */
7 add sp, sp, #8 /* gibt den Speicher auf dem Stack wieder frei */

```

Werden weiter Unterprogramme aufgerufen, muss auch die Rücksprungadresse lr auf dem Stack gesichert werden, sonst geht diese beim Prozeduraufruf verloren.

Kurzschreibweisen (Pseudoinstruktionen) für das reservieren/freigeben von Speicher und dem ablegen/wiederherstellen von Wörtern sind die Instruktionen:

```

1 push {lr} /* Ablegen der Ruecksprungadresse */
2 ...
3 pop {lr} /* Wiederherstellen der Ruecksprungadresse */

```

5.2 Rekursion

Bei einer rekursiven Programmausführung repräsentiert die Anzahl an Rücksprungadressen auf dem Stack im wie vielen Aufruf wir uns befinden. Der Ablauf eines Unterprogramms wird auch als Inkarnation (wiederholter Unterprogrammaufruf) bezeichnet.

Auszug einer rekursiven Funktion:

```

1 .text /* markiert den Beginn eines Code Segments */
2 fak:   push {r0, lr}
3       cmp r0, #1 /* Rekursionsende pruefen */
4       blt else /* branch less ⇔ N = 1 */
5       sub r0, r0, #1 /* n-1 */
6       bl fak /* rekursiver Funktionsaufruf */
7       /* Ruecksprungadresse beim zurueckgehen */
8
9 RA_2:  ldr r1, [sp, #4] /* laden von n */
10      mul r0, r1, r0 /* fak(n-1) * n */
11 fin:  pop {lr} /* laden der Rueckkehradresse */
12      add sp, sp, #4 /* Freigeben des Stackspeichers */
13      bx lr /* Sprung zum Aufrufer */
14
15 else:  mov r, #1 /* Rekursionsanker */
16      b fin /* Rekursionsende, abbauen des Stacks */

```

6 Vorlesung 6

6.1 OpenMP

OpenMP ist eine Sammlung von Compiler-Anweisungen (Direktiven) und Bibliotheksfunktionen für die Thread Programmierung.

Es ist eine Kombination von C, C++ und Fortran. Dadurch wird threadparalleles Arbeiten ermöglicht.

OpenMP startet als einzelner Thread und forkt für einzelne parallele Programmabschnitte zusätzliche Threads. Nach Abschluss joinen die Threads wieder zusammen. Dies wird auch **Fork-Join-Programmiermodell** bezeichnet.

```
1 #include <omp.h> /* OpenMP Library */
2 ...
3 #pragma omp parallel for private(x) reduction(+:sum)
4 for( i = 0; i < num_steps; i++) {
5     x = (i + 0.5) * step;
6     sum += 4.0 / (1.0 + x * x)
7 }
8 ...
```

6.2 Assembler

Der Assembler ist ein Programm, dass die Aufgabe hat, Assemblerbefehle in Maschinencode zu transformieren und dabei symbolische Namen Maschinenadressen zuweist und eine oder mehrere Objektdatei(en) erzeugt.

Varianten sind Crossassembler, welche Maschinencode für eine andere Plattform erzeugen als sie gerade laufen, sowie Disassembler welche Maschinensprache in Assemblersprache übersetzen.

Die Übersetzung erfolgt in zwei Schritten:

1. Schritt:

- Auffinden von Marken um Beziehungen zwischen symbolischen Namen und Adressen zu kennen (Syntax- und Kontextanalyse)
- Übersetzen jedes Assemblerbefehls durch Kombination der Opcodes⁵, Bezeichner und Marken zu legalen Instruktionen (Codegen)

2. Schritt: Erzeugen einer oder mehrerer Objektdateien. Diese ist meist nicht ausführbar, da sie auf Funktionalitäten anderer Dateien angewiesen ist.

Probleme bei den Schritten:

- Bei Schritt 1 sind zukünftige Marken nicht bekannt → Aufteilen in Syntax- und Kontextanalyse (2 Läufe)
- Bei Schritt 2 werden in der Objektdatei absolute Adressen verwendet
 - Programm kann direkt ausgeführt werden, aber der Speicherort muss vorher bekannt sein
 - Nachträgliches verschieben des Programms ist nicht möglich

⁵Nummer des entsprechenden Maschinenbefehls

- Bei Schritt 2 werden relative Adressen verwendet, und die aktuelle Adresse bei Programmbeginn übergeben
 - Erzeugt ggf. mehrere Objektdateien
 - Adressen werden relativ zu den einzelnen Objektdateien vergeben
 - Vor Ausführung sind weitere Transformationsschritte nötig, Aufgabe des Binders/Linkers und Laders

6.3 Objekt-Programme

Wir unterscheiden:

- Relocatable object files: Enthält binären Code und Daten, sodass diese mit anderen relocatable object files zu einem ausführbaren Objektfile zusammengefügt werden können
- Executable object files: Enthält binären Code und Daten, welche direkt in den Speicher kopiert und ausgeführt werden können
- Shared object files: Spezialfall der relocatable object files, welche in den Speicher geladen werden können und dynamisch mit anderen Objekt-Files zusammengeführt werden können

In der Regel generieren Compiler und Assembler relocatable object files.

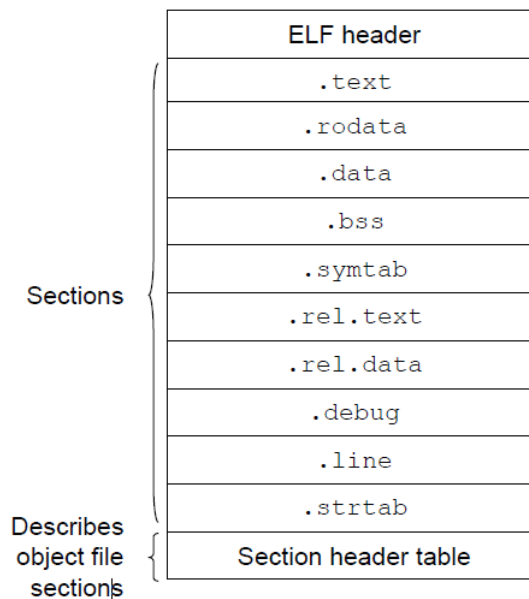


Abbildung 6: Das Format eines typischen ELF relocatable object files

Ein ELF⁶ relocatable object file beginnt mit einer 16-Byte Sequenz:

- Wort-Größe
- Byte-Ordering
- Weitere Infos für den Binder/Linker z.B. Maschinentyp

Die einzelnen Segmente des files haben folgende Bedeutung:

- .text Maschinencode des compilierten/assemblierten Programms

⁶ELF - Executable and Linkable Format

- `.rodata` Daten, welche nur gelesen werden müssen z.B. Formatierungsstrings oder Sprungtabellen für `switch`
- `.data` Initialisierte globale Variablen
- `.bss`⁷ Uninitialisierte globale Variablen
- `.symtab` Symboltabelle mit Infos über Funktionen und globale Variablen
- `.rel.text` Liste an Stellen, welche beim Linken modifiziert werden müssen, z.B. Kombination mit anderen files
- `.rel.data` Relocations Informationen für globale Variablen
- `.debug` Debugging Symboltabelle (Nur erzeugt, falls C-Compiler mit `-g` aufgerufen wurde)
- `.line` Zuordnung der C-Anweisung zu Maschinencode (Nur falls `-g` gesetzt)
- `.strtab` Zeichentabelle für Symboltabelle und Debugging Symboltabelle

Diese Datei kann mit dem Befehl **`readelf -h filename.o`** geöffnet werden.

Mit **`readelf -a filename.o`** erhält man eine Übersicht über die wichtigsten Einträge.

Mit **`objdump -S filename.o`** erhält man den Maschinencode.

Analysewerkzeuge sind:

- IDA, ein kommerzieller Disassembler
- Ghidra: Reverse-Engineering-Werkzeug der NSA

6.4 Binder und Lader

Der Binder (linker) hat die Aufgabe aus mehreren einzelnen verschiebbaren Objekt files ein ausführbares Objektprogramm zu erzeugen, indem die noch offenen externen Referenzen aufgelöst werden.

Das Objektprogramm kann dann durch einen Lader zur Ausführung gebracht werden.

Ein Lader (loader) ist ein Systemprogramm, welches Objektprogramme in den Speicher lädt und ggf. deren Ausführung anstößt.

Der Lader lädt ein Programmmodul (Lademodul) beginnend mit einer vom Betriebssystem vorgegebenen Startadresse in den Hauptspeicher.

Varianten sind:

- absolutes Laden
- relatives Laden
- dynamisches Laden zu Laufzeit

6.5 Laufzeit

Laufzeitmessungen von Programmen auf Assemblerebene können mit sogenanntem Programm Profiling durchgeführt werden.

Mit **`gcc -pg -o functionname programmname.c`** kann eine Objektdaten erzeugt werden, welche ein solches Profiling durchführt. Die Datei wird mit **`./functionname`** ausgeführt, dann lässt man sich mit **`gprof functionname`** die Profile Datei ausgeben.

⁷Block Storage Start, Better Save Space

Dies gibt Aufschluss über eventuelle Bottlenecks und zeigt Informationen wie z.B. die Zeit die das Programm in einzelnen Subroutinen verbraucht hat. Aufbauend darauf kann dann z.B. Loop-Unrolling oder Thread Programmierung durchgeführt werden.