Rechnerorganisation
Sommersemester 2023 – 6. Vorlesung

Prof. Stefan Roth, Ph.D.

Technische Universität Darmstadt

5. Juni 2023

Inhalt

Ankündigungen und Informationen

Ankündigungen und Informationen

- Anmeldung zur Studienleistung
  - Auf der Webseite des FB Informatik steht zum Thema Anmeldung zur Studienleistung Folgendes (siehe auch Vorlesung 1, Folie 11):

    Hinweis: Studienleistungen werden häufig als Teilleistungen erbracht. Studierende sollten sich vor dem Erbringen möglicher Teilleistungen zur Studienleistung des entsprechenden Moduls anmelden. Ein Anrecht auf Wiederholung von Teilleistungen bei verspäteter Anmeldung besteht nicht!
  - Daran haben wir uns gehalten...
  - Laut Studienbüro fällt RO aber in eine LV-Kategorie, bei der dies nicht erforderlich ist, d.h. wir als Organisatoren müssen dies nicht fordern und dürfen auch spätere Anmeldungen berücksichtigen.
  - Ergo: Alle abgelegten Testate sind gültig, egal ob vorher angemeldet oder nicht 
  - Dennoch ist es nötig sich für die Studienleistung anzumelden!

Ankündigungen und Informationen

- 1. Programmiertestat läuft noch bis zum 11. Juni 2023.
- Die voraussichtlichen Daten für die nächsten 3 Testate sind online:
  - 3. Übungstestat: 19. Juni – 26. Juni 2023
  - 2. Programmiertestat: 26. Juni – 03. Juli 2023
  - 4. Übungstestat: 03. Juli – 10. Juli 2023
- Diese Woche ist am Donnerstag Feiertag.
  Studierende, die ihre Übung diese Woche am Donnerstag haben, können in eine andere Übung gehen (Feiertag).
- Da es letzte Woche keine neue Übung gab, kann die Übungsstunde diese Woche dafür genutzt werden bereits behandeltes Material zu verfestigen.
- Erinnerung: Jeder der das Programmiertestat für die Studienleistung angerechnet haben will, muss sich bei einem Termin mit einem/er der Tutoren/innen anmelden.

Interessantes aus den Medien

- Heise, 27. Mai 2023: Nur 512 Byte: Dieser C-Compiler passt in einen Bootsektor
  - Untermenge von C
  - IA16-Assembler
  - Blog: `https://xorvoid.com/sectorc.html`
  - GitHub: `https://github.com/xorvoid/sectorc`

  - Maschinencode (Base64 Encoding):


```
6gUAwAdoADAfaAAgBzH/6DABPfQYdQXoJQHr8+gjAVOJP+gSALDDqluB+9lQdeAG/zdoAEAfy+gI
AegFAYnYg/hNdFuE9nQNsOiqiwcp+IPoAqvr4j3/FXUG6OUAquvXPVgYdQXoJgDrGj0C2nUGV+gb
AOsF6CgA68Ow6apYKfiD6AKrifgp8CaJRP7rrOg4ALiFwKu4D4Srq1fonP9ewz2N/HUV6JoA6BkA
ieu4iQRQuIs26IAAWKvD6AcAieu4iQbrc4nd6HkA6HYA6DgAHg4fvq8Bra052HQGhcB19h/DrVCw
UKroWQDoGwC4WZGrW4D/wHUMuDnIq7i4AKu4AA+ridirH8M9jfx1COgzALiLBOucg/j4dQXorf/r
JIP49nUI6BwAuI0G6wyE0nQFsLiq6wa4iwarAduJ2KvrA+gAAOhLADwgfvkx2zHJPDkPnsI8IH4S
weEIiMFr2wqD6DABw+gqAOvqicg9Ly90Dj0qL3QSPSkoD5TGidjD6BAAPAp1+eu86Ln/g/jDdfjr
slIx9osEMQQ8O3QUuAACMdLNFIDkgHX0PDt1BIkEMcBaw/v/A8H9/yvB+v/34fb/I8FMAAvBLgAz
wYQA0+CaANP4jwCUwHf/lcAMAJzADgCfwIUAnsCZAJ3AAAAAAAAAAAAAAAAAAAAAAAAAAAAAVao=
```

Compilieren, Assemblieren und Linken

Ein erstes Assemblerprogramm – noch in C

```
1  /* Addition */
2  #include <stdio.h>
3
4  int main()
5  {
6    int p = 5;
7    int q = 12;
8    int result = p + q;
9    printf("result␣ist␣%d␣\n",result);
10   return 0;
11 }
```

- `gcc addition.c` übersetzt das C-Programm[1]
- `gcc -S addition.c` generiert das Assemblerprogramm

[1] Hinweis: Auf `clientssh-arm` verwenden wir einen Cross-Compiler und nicht den systemeigenen `gcc`.

Ein erstes Assemblerprogramm – addition.s – ARM® Architektur

```
1           .file   "addition.c"
2           .section        .rodata
3           .align  2
4   .LC0:
5           .ascii  "result_ist_%d_\012\000"
6           .text
7           .align  2
8           .global main
9           .syntax unified
10          .arm
11          .fpu vfp
12          .type   main, %function
13  main:
14          @ args = 0, pretend = 0, frame = 16
15          @ frame_needed = 1, uses_anonymous_args = 0
16          push    {fp, lr}
17          add     fp, sp, #4
18          sub     sp, sp, #16
19          mov     r3, #5
20          str     r3, [fp, #-8]
21          mov     r3, #12
22          str     r3, [fp, #-12]
23          ldr     r2, [fp, #-8]
24          ldr     r3, [fp, #-12]
25          add     r3, r2, r3
26          str     r3, [fp, #-16]
27          ldr     r1, [fp, #-16]
28          ldr     r0, .L3
29          bl      printf
30          [...]
```

Ein zweites Assemblerprogramm – addition.s – ARM® Architektur

```
1  /* first.s */
2  .global main /* Einsprungpunkt Hauptprogramm */
3
4  main:              /* Hauptprogramm */
5      mov r1, #5     /* Schreibe eine 5 in das Register r1 */
6      mov r2, #12
7      add r0, r1, r2 /* Addiere r1 und r2,Ergebnis in r0 */
8      bx lr          /* Springe zurueck zum aufrufenden Programm */
```

Ein erstes Assemblerprogramm – noch in C

```
1  /* Addition */
2  #include <stdio.h>
3
4  int main()
5  {
6    int p = 5;
7    int q = 12;
8    int result = p + q;
9    printf("result ist %d \n",result);
10   return 0;
11 }
```

- `gcc addition.c` übersetzt das C-Programm
- `gcc -S -O1 addition.c` generiert das Assemblerprogramm

Analyse des Assemblerprogramms – addition.s – ARM® Architektur

```
1  main:
2          @ args = 0, pretend = 0, frame = 0
3          @ frame_needed = 0, uses_anonymous_args = 0
4          push    {r4, lr}
5          mov     r1, #17
6          ldr     r0, .L3
7          bl      printf
8          mov     r0, #0
9          pop     {r4, pc}
10 .L4:
11         .align  2
12 .L3:
13         .word   .LC0
14         .size   main, .-main
15         .section        .rodata.str1.4,"aMS",%progbits,1
16         .align  2
17 .LC0:
18         .ascii  "result␣ist␣%d␣\012\000"
19         .ident  "GCC:␣(Raspbian␣6.3.0-18+rpi1+deb9u1)␣6.3.0␣20170516"
```

Diskussion der Assemblerprogramme

- Die Übersetzung eines Programms aus einer Hochsprache nach Assembler kann offenbar sehr verschiedene Ergebnisse haben.
- Nicht alle in Hochsprachen sichtbare Sprachelemente werden in Assembler abgebildet.
- Beispiele:
  - Nicht jede lokale Variable wird auf dem Stack sichtbar. Ggf. Realisierung von lokalen Variablen bei Schleifen ausschließlich in Registern.[2]
  - Bei konstanten Variablen (=Konstanten) wird ggf. eine sogenannte **Common subexpression elimination** durchgeführt. Das bedeutet, dass Teilausdrücke zur Übersetzungszeit berechnet werden und als Konstanten gespeichert werden.
  - ...

[2] Dies hängt auch davon ab, mit welcher Optimierungseinstellung ein Compiler aufgerufen wird. Beim gcc sind das z. B. (-O1, -O2).

π Berechnung mit Trapezintegration

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main (void)
5  {
6    long i;
7    long num_steps = 2000000000;
8    double x, pi, step, sum = 0.0;
9    step = 1.0 / (double) num_steps;
10   for (i = 1; i <= num_steps; i++)
11   {
12     x = (i - 0.5) * step;
13     sum = sum + 4.0 / (1.0 + x * x);
14   }
15   pi = step * sum;
16   printf("PI_%lf\n", pi);
17  return 0;
18 }
```

Laufzeiten des Beispielprogramms I

- Raspberry Pi, mit ARM® Cortex-A53, 4 Kerne, 1 GB RAM

Variante	Optimierungsstufe	Laufzeit
1	–	1m34,416s
2	-O1	1m5,943s
3	-O2	1m4,409s

- **Achtung**: Raspberry Pi nicht zu lange laufen lassen. Überhitzungsgefahr!

Laufzeiten des Beispielprogramms II

- clientshh-arm, ThunderX, 48 Kerne, 2.5 GHz, 64 GB RAM,

Variante	Optimierungsstufe	Laufzeit
1	–	1m15,076s
2	-O1	0m50,051s
3	-O2	0m46,048s

- Parallelisierung kann dies weiter beschleunigen, s. Vorlesung Parallele Programmierung

Laufzeiten des Beispielprogramms III

- MacBookPro, M1 Pro, 8+2 Kerne, 2.06 – 3.22 GHz, 32GB RAM

Variante	Optimierungsstufe	Laufzeit
1	–	0m06,992s
2	-O1	0m01,915s
3	-O2	0m01,946s

- Hinweis: Messung mit ARM64 code.

Compiler, Assembler, Linker, Loader
Phasen der Übersetzung

- Ausgangspunkt: C Programm
- Zur Ausführung auf dem Rechnersystem muss es in Maschinencode übersetzt werden.
- Beispiel: Unix System: `gcc -o hello hello.c`

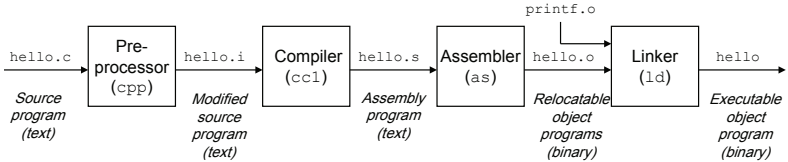


Abbildung: Quelle: [BO10, S. 39]

Compiler, Assembler, Linker, Loader
Phasen der Übersetzung

- Übersetzungsvorgang ist in verschiedene Phasen unterteilt
- 1. Phase (Preprocessor)
  - Aufbereitung durch Ausführung von Direktiven (mit #)
  - Z. B. Bearbeiten von `#include <stdio.h>`
    - Lesen des Inhalts der Datei `stdio.h`
    - Kopieren des Inhalts in die Programmdatei
  - Ausgabe: C-Programm mit der Endung .i
- 2. Phase (Compiler)
  - Übersetzt das C-Programm `hello.i`
  - in ein Assemblerprogramm `hello.s`
- 3. Phase (Assembler)
  - Übersetzt hello.s in Maschinensprache
  - Ergebnis ist das Objekt-Programm `hello.o`

Compiler, Assembler, Linker, Loader
Phasen der Übersetzung

- 4. Phase (Linker/Binder)
  - Zusammenfügen verschiedener Module
    - Beispielprogramm nutzt die `printf` Funktion
    - Der Code von `printf` existiert bereits übersetzt in einer Bibliothek (der Standard C Library) als `printf.o` Datei
  - Der Linker (Binder) kombiniert `hello.o` und `printf.o` zu einem ausführbaren Programm (u. a. Auflösen von Referenzen)
  - Ausgabe des Bindevorgangs: **`hello`** Datei
  - `hello` ist eine ausführbare Objekt-Datei, die in den Speicher geladen und ausgeführt werden kann

Aufgabe & Funktionsweise eines Assemblers

Aufgabe & Funktionsweise eines Assemblers I

- Assembler übersetzt Assemblerprogramm (Mnemonics) in Maschinensprache.
- Allgemeinere Definition ist: Ein Assembler ist ein Programm, das die Aufgabe hat, Assemblerbefehle in Maschinencode zu transformieren und dabei
  - symbolischen Namen (z. B. Labels bei Sprungmarken) Maschinenadressen zu zuweisen
  - und eine oder mehrere Objektdatei(en) zu erzeugen.
- Assembler gibt es außerdem als
  - **Crossassembler**: Assembler läuft auf einem Rechnersystem X, generiert aber Maschinencode für eine Plattform Y. Kommt sehr häufig im Bereich der Embedded Systems vor (Intel IA32 als Entwicklungsplattform $\rightarrow$ ARM als Zielplattform).
  - **Disassembler**: Übersetzung von Maschinensprache in Assemblersprache $\rightarrow$ üblicherweise Verlust von Kommentaren. und symbolischen Namen

Aufgabe & Funktionsweise eines Assemblers II

- Assembler übersetzt Datei in Assemblersprache in Datei mit binären Maschinencode in zwei Schritten.
- 1. Schritt:
  - Auffinden von Speicherpositionen mit Marken, so dass Beziehungen zwischen symbolischen Namen und Adressen bei Übersetzung von Instruktionen bekannt sind.
  - Übersetzung jedes Assemblerprogrammbefehls durch Kombination der numerischen Äquivalente der Opcodes, Registerbezeichner und Marken in eine legale Instruktion.
- 2. Schritt: Assembler erzeugt ein oder mehrere Objektdateien, die Maschinencode, Daten und Verwaltungsinformationen enthält. Eine Objektdatei ist meist nicht ausführbar, da im Allgemeinen auf Funktionen, Prozeduren oder Daten in anderen Dateien verwiesen wird.

Aufgabe & Funktionsweise eines Assemblers III

- Probleme beim 1. Schritt: Nutzen von Marken bevor sie definiert sind
- Beispiel: Abarbeiten des Programms Zeile für Zeile, Befehle können Vorwärts-Referenzen enthalten

```
1       cmp r0, r1       /* setze Flags auf Basis der */
2                        /* Rechnung r0 - r1 = -4 */
3                        /* N=1, da Ergebnis negativ */
4                        /* Statusflags: N=1, Z=0, C=0, V=0 */
5       beq there        /* kein Sprung (Z != 1) */
6       orr r1, r1, #1   /* r1 = r1 or 1 = 9 */
7   there:
8       add r1, r1, #78 /* r1 = r1 + 78 = 87 */
```

- Problem: Adresse von Sprungmarke **there:** ist beim ersten Auftreten unbekannt $\rightarrow$ *korrekter* Code kann nicht erzeugt werden
- Lösung: Assembler macht 2 Läufe (two-pass) über das Programm
  - 1-ter Lauf (Phase): Zuordnen von Maschinenadressen
  - 2-ter Lauf (Phase): Erzeugen des Codes

Aufgabe & Funktionsweise eines Assemblers IV

- Probleme beim 2. Schritt $\rightarrow$ Erzeugen der Objektdatei (.o-Datei)
- 1. Fall:
  - Assembler verwendet **absolute** Adressen und eine Objektdatei (Objektprogramm)
  - dann ist das Laden unmittelbar möglich, aber absolut adressiert bedeutet, der Speicherort muss vorher bekannt sein
  - Nachteil $\rightarrow$ Verschieben des Programms im Speicher ist nicht möglich
- 2. Fall:
  - Assembler verwendet **relative** Adressen und als Eingabe ggf. mehrere Programm-Segmente
  - Assembler-Ausgabe: ≥ 1 Objekt-Datei(en)
  - Adressen werden relativ zu den einzelnen Objektdateien vergeben
  - Konsequenz daraus $\rightarrow$ weitere Transformationsschritte sind notwendig
    $\rightarrow$ Aufgabe des Linkers (Binders) und Loaders (Laders)

Aufbau eines Objekt-Programms

Aufbau des Objekt-Programms I

- Objekt-Programme (im Folgenden Files) kommen in drei Formen vor [BO10, S. 691]
  - Relocatable[3] object files: Enthalten binären Code und Daten in einer Form, die mit anderen verschiebbaren Objekt-Files zu einem ausführbaren Objekt-File zusammengefügt werden können.
  - Executable object files: Enthalten binären Code und Daten in einer Form, die direkt in den Speicher kopiert und ausgeführt werden können.
  - Shared object files: Spezieller Typ von *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*.

[3] verschiebbare

Aufbau des Objekt-Programms II

- Format eines typischen ELF[4] *relocatable object files*

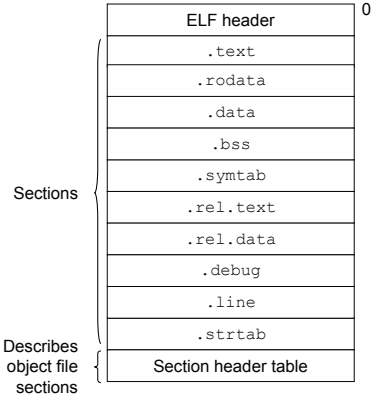


Abbildung: Quelle: [BO10, S. 692]

[4] Executable and Linkable Format

Aufbau des Objekt-Programms III – Bedeutung der Felder

- ELF *relocatable object files* beginnt mit einer 16-Byte Sequenz
  - Wort-Größe
  - Byte-Ordering (Little Endian, Big Endian)
  - Weitere Informationen, die vom Linker verarbeitet werden, z. B. Maschinentyp (ARM, IA32)
- **.text**
  Maschinencode des compilierten/assemblierten Programms
- **.rodata**
  Daten, die nur gelesen werden müssen, z. B. Formatierungsstrings in Ausgabefunktionen oder Sprungtabellen in switch-Statements
- **.data**
  Initialisierte globale Variablen
- **.bss**[5]
  Uninitialisierte globale Variablen $\rightarrow$ belegt kein Platz im Objekt-File

[5] Block Storage Start, Better Save Space

Aufbau des Objekt-Programms IV – Bedeutung der Felder

- **.symtab**
  Symboltabelle mit Informationen über Funktionen und globale Variablen, die im Programm definiert bzw. referenziert sind.
- **.rel.text**
  Eine Liste von Stellen im .text Segment, welche beim Linken modifiziert werden müssen (z. B. kombinieren mit anderen Objekt-Files).
- **.rel.data**
  Relocations Informationen für globale Variablen
- **.debug**
  Debugging Symboltabelle → wird nur erzeugt, wenn C-Compiler mit der Option -g aufgerufen wird.
- **.line**
  Zuordnung der C-Anweisungen zu Maschinencode → wird nur erzeugt, wenn C-Compiler mit der Option -g aufgerufen wird.
- **.strtab**
  Zeichentabelle für Symboltabelle und Debugging Symboltabelle

Beispiel – Schleife.s

```
1  /* -- Schleife.s */
2  .global main /* Einsprungpunkt Hauptprogramm */
3
4  main:              /* Hauptprogramm */
5      mov r0, #1     /* r0 = pow = 1 */
6      mov r1, #0     /* r1 = x = 0 */
7  WHILE:
8      cmp r0, #256
9      beq DONE
10     lsl r0, r0, #1 /* pow = pow * 2 */
11     add r1, r1, #1 /* x = x + 1 */
12     b WHILE
13 DONE:
14     mov r0, r1
15     bx lr          /* zurueck zum Aufrufer */
```

Beispiel – Objekt-Programm Schleife.o

- Übersetzung dieses Programms erfolgt mit **as -o Schleife.o Schleife.s**
- Lesen Header ELF *relocatable object files* **readelf -h Schleife.o**

```
ELF Header:
 Class:                             ELF32
 Data:                              2's complement little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              REL (Relocatable file)
 Machine:                           ARM
 Version:                           0x1
 Entry point address:               0x0
 Start of program headers:          0 (bytes into file)
 Start of section headers:          348 (bytes into file)
```

Beispiel – Objekt-Programm Schleife.o

- Mit **readelf -a Schleife.o** erhält man eine Übersicht über die wichtigsten Einträge
- Mit **objdump -S Schleife.o** erhält man den Maschinencode

```
Schleife.o:     file format elf32-littlearm
Disassembly of section .text:
00000000 <main>:
   0: e3a00001  mov r0, #1
   4: e3a01000  mov r1, #0
00000008 <WHILE>:
   8: e3500c01  cmp r0, #256 ; 0x100
   c: 0a000002  beq 1c <DONE>
  10: e1a00080  lsl r0, r0, #1
  14: e2811001  add r1, r1, #1
  18: eafffffa  b 8 <WHILE>
0000001c <DONE>:
  1c: e1a00001  mov r0, r1
  20: e12fff1e  bx lr
```

Analysewerkzeuge[7]

- gcc Toolchain
- IDA[6] ist ein Disassembler, der es ermöglicht, Binärcode in Assemblersprache umzuwandeln.
- Ghidra: Reverse-Engineering-Werkzeug der NSA (Open-Source) - Link
- Radare2: Radare2 ist ein Framework für Reverse Engineering und die Analyse von Binärdateien.

[6] Interactive Disassembler
[7] auch Reverse-Engineering-Werkzeug

Linker & Loader

Linker & Loader

Linker/Binder

- Der Binder[a] hat die Aufgabe, aus einer Menge von einzelnen verschiebbaren Objekt-Files ein ausführbares Objektprogramm zu erzeugen, indem die noch offenen externen Referenzen aufgelöst werden.
- Das Objektprogramm kann durch einen Lader zur Ausführung gebracht werden.

[a] engl. linker

Loader/Lader

- Ein Lader[a] ist ein Systemprogramm, das die Aufgabe hat, Objektprogramme in den Speicher zu laden und ggf. deren Ausführung anzustoßen.
- Dazu wird das Objektprogramm in den Speicher kopiert.

[a] engl. loader

Arbeitsweise eines Loaders

- Aufgabe des Loaders (vgl. [BO10, S. 713 f.])
  - ein Programmmodul (Lademodul) wird – beginnend mit einer vom Betriebssystem vorgegebenen Startadresse – in den Hauptspeicher geladen
- Varianten des Ladevorgangs
  - absolutes Laden (Absolute Loading)
  - relatives Laden (Relocatable Loading)
  - dynamisches Laden zur Laufzeit (Dynamic Run-Time Loading)

Laufzeitanalyse von Programmen

Laufzeitanalyse von Programmen

Laufzeitanalyse von Programmen

- Für das Schreiben von effizienten Programmen muss man wissen, wo die *Performance Bottlenecks* sind
- Für heute → Performance heißt Ausführungszeit des Programms
- Andere Größen der Optimierung
  - Speichernutzung
  - Programmgröße
- Unter Unix (und Windows) sind verschiedene Programme zur Performance Analyse vorhanden

Analyse C Programm mit zwei Funktionen – Unterprogramme

```
1  #include <stdio.h>
2  long fak_ite(int n) {
3    int i;
4    long result = 1;
5    for (i = 2; i <= n; i++)
6      result *= i; /* (result = result * i) */
7    return result;
8  }
9  long fak_rek(int n) {
10   if (n>=1)
11     return fak_rek(n-1) * n;
12   else
13     return 1;
14 }
```

Analyse C Programm mit zwei Funktionen – Hauptprogramm

```
1  int main(void) {
2    int  n = 30;
3    long erg1, erg2;
4    long schleife;
5    for (schleife = 1; schleife < 10000000; schleife++) {
6     erg1 = fak_ite(n);
7     printf("Fakultaet: %d\n", erg1);
8     erg2 = fak_rek(n);
9     printf("Fakultaet: %d\n:", erg2);
10   }
11   return 0;
12 }
```

Analyse von C Programmen – Diskussion

- Zur Erinnerung: *schleife* ist in C eine lokale Variable des Hauptprogramms
- Das Hauptprogramm ist eigentlich auch ein **Unterprogramm**
- Lokale Variablen werden in Unterprogrammen auf dem Stack/bzw. durch Register realisiert.
- **Zur Erinnerung**: Anzahl Register sind beschränkt, Operationen (z. B. **add**) sind nur Registern auszuführen. Transfer von Registerinhalten auf Stack (und zurück) kostet Zeit
- Ausführungszeit (using time, gcc) auf dem Raspberry Pi
  - ca. 8 s (ohne Optimierung)
  - $<$ 1 s (-O1)

Analyse von C Programmen – gprof

- Für das sogenannte *Programm Profiling* stehen verschiedene Möglichkeiten zur Verfügung (vgl. [BO10] [S. 574])
- Im Beispiel: Hauptprogramm und zwei Funktionen (`fak_ite` und `fak_rek`)
- Welche Zeit wird im Hauptprogramm bzw. in den beiden Funktionen verbracht?
- gcc (und andere Compiler) können darüber Aussagen liefern
  - `gcc -pg -o funktion_fak funktion_fak.c` (-pg ist ein run-time flag)
  - `./funktion_fak` (Aufruf des Programms, Programm läuft langsamer (ca. 25 s)
  - `gprof funktion_fak gmon.out > analysis.txt` (wertet die Profile-Datei aus)

Analyse eines C Programms mit zwei Funktionen, Zeiten

- Aufsplittung der Zeiten für das Hauptprogramm und die beiden Funktionen

```
Each sample counts as 0.01 seconds.
 %   cumulative   self              self     total
time   seconds   seconds    calls ns/call ns/call name
56.30      4.51     4.51  9999999  451.00  451.00 fak_rek
41.32      7.82     3.31  9999999  331.00  331.00 fak_ite
 2.37      8.01     0.19                           main
```

- Suchen (und finden) von Funktionen, die viel Zeit brauchen → ggf. Optimierungen
  - z. B. Loop-Unrolling
  - oder bessere (parallele) Algorithmen
  - Thread Programmierung

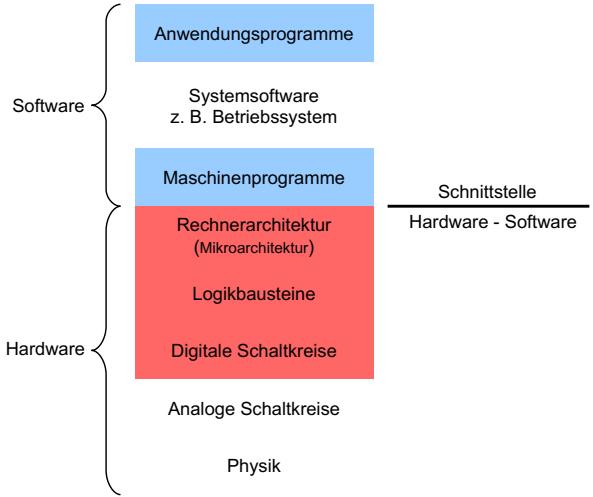
Mikroarchitekturen von Rechnersystemen

Lernziele und Lerninhalte der Rechnerorganisation

Quelle: Modulhandbuch

- Architektur von Mikroprozessoren: Programmierung in Assembler- und Maschinensprache, Adressierungsarten, Werkzeugflüsse, Laufzeitumgebung
- Mikroarchitektur: Befehlssatz und architektureller Zustand, Leistungsbewertung, Mikroarchitekturen mit Eintakt-/Mehrtakt-/Pipeline-Ausführung, Ausnahmebehandlung, fortgeschrittene Mikroarchitekturen
- Speicher und Ein-/Ausgabesysteme: Leistungsbewertung, Caches, virtueller Speicher, Ein-/Ausgabetechniken, Standardschnittstellen

Schichtenmodell eines Computers



Komponenten eines Rechnersystems

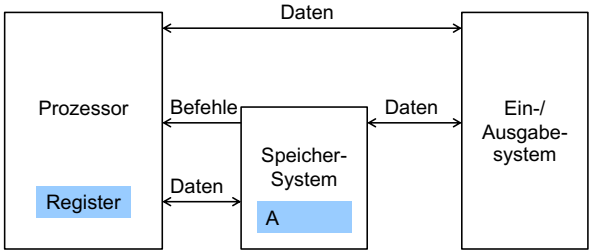
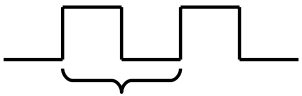


Abbildung: Komponenten eines Rechnersystems (verfeinerte Darstellung)

Takt und Taktfrequenz

- Die Komponenten eines Rechnersystems müssen in der Regel eine gemeinsame Zeitbasis haben, damit das Zusammenspiel funktioniert. Diese Zeitbasis nennt man auch Takt/Taktsignal/Systemtakt.
- Der Takt ist ein Rechtecksignal.

- Er dient der Synchronisation der Komponenten eines Rechnersystems, deswegen auch Systemtakt.
- Aus dem Rechtecksignal kann man die Periodendauer/die Taktfrequenz ableiten ($f = \frac{1}{T}$).



- Schluss: je höher die Taktfrequenz ist, desto schneller verarbeitet das Rechnersystem Daten.

Interpretation und Ausführung eines Befehls

- Wie funktioniert die Verarbeitung eines Befehls bzw. der Daten in einem Rechnersystem?
  - Sowohl der Befehl, als auch die Daten stehen in dem Speicher(-system).
- Die Aufgabe des Prozessors bzw. des Steuerwerks ist es, die Befehle aus dem Speicher zu lesen (bzw. die Anweisungen dafür zu geben). Diesen Vorgang bezeichnet man als **Befehlsholphase**.[8]
- Wenn der Befehl geholt ist und in einem Register steht, muss er dekodiert werden. Diesen Vorgang nennt man **Befehlsdekodierung**.[9]
- Als letztes wird der Befehl ausgeführt (**Befehlsausführung**).[10] Danach wird der nächste Befehl aus dem Speicher geholt.
- Man spricht auch von den *drei Phasen der Befehlsausführung*.

[8] engl.: instruction fetch
[9] engl.: instruction decode
[10] engl.: instruction execute

Terminologie I

- Im Kontext von **Rechnerarchitekturen/Prozessorarchitekturen** gibt es einige Begriffe, die im Folgenden erklärt werden.

ISA	instruction set architecture (Menge der verfügbaren Instruktionen/Befehle)
RISC	reduced instruction set computer (kleine und schnell ausführbare ISA), Beispiel ARM®
CISC	complex instruction set computer (schwergewichtige ISA), Beispiel Intel®
SIMD	single instruction multiple data (vector processing), Beispiel NEON beim ARM®
VLIW	very long instruction word (static multi-issue), superscalar processors (dynamic multi-issue)

Terminologie II

- Im Kontext von **Rechnerarchitekturen/Prozessorarchitekturen** gibt es einige Begriffe, die im Folgenden erklärt werden.

μarch microarchitecture (ISA Implementierung)

- Anzahl der Instruktionen/Befehle pro Zyklus (IPC[11])
- Pipelining für Instruktionslevel Parallelismus (ILP[12])
- Sprungvorhersage (engl.: branch prediction)
- out-of-order execution von Befehlen (engl.: dynamic instruction scheduling)
- multi-issue systems (engl.: start multiple operations per cycle)

[11] engl.: instructions per cycle
[12] engl.: instruction level parallelism

Mikroarchitektur [HH16, S. 385 - 484]

- Einführung in die Mikroarchitektur
- redEintakt-Prozessor
- Mehrtakt-Prozessor
- Pipeline-Prozessor
- Analyse der Rechenleistung
- Ausnahmebehandlung
- Weiterführende Themen

Einführung in die Mikroarchitektur

Mikroarchitektur

- Hardware-Implementierung einer Architektur

Prozessor/Rechnersystem

- **Datenpfad**: verbindet funktionale Blöcke
- **Kontrollpfad**: Steuersignale/Steuerwerk

Mikroarchitektur

- Mehrere Implementierungen für eine Architektur (hier ARM®) möglich
- Unterscheidung ist der Takt bzw. die Art der Befehlsausführung
- Wie wird ein Befehl[13], z. B. add, ausgeführt?

Eintakt-Implementierung[a]

[a] Im Folgenden wird der Begriff Eintakt-Prozessor verwendet.

- Jeder Befehl wird in einem Takt ausgeführt.

Mehrtakt-Implementierung

- Jeder Befehl wird in Teilschritte zerlegt.

Pipelined-Implementierung

- Jeder Befehl wird in Teilschritte zerlegt. Mehrere Teilschritte werden gleichzeitig (parallel) ausgeführt.

[13] Instruktion, Maschinenbefehl

Zusammenfassung und Ausblick

Zusammenfassung und Ausblick

Zusammenfassung

- Compilieren, Assemblieren und Linken
- Laufzeitanalyse von Programmen
- Mikroarchitekturen von Rechnersystemen

Ausblick

- Mikroarchitekturen von Rechnersystemen
- Mikroarchitekturen eines ARM-Prozessors

Lernkontrolle

- Die verschiedenen Phasen einer Programmübersetzung habe ich verstanden und kann die notwendigen Programme benennen ☑
- Den Aufbau von Objektprogrammen kann ich nachvollziehen ☑
- Die Werkzeuge der Programmanalyse habe ich kennengelernt und habe den Nutzen verstanden ☑
- Ich habe die Terminologie im Kontext von Rechnerarchitekturen/Prozessorarchitekturen verstanden und kann diese richtig nutzen ☑
- ...

Literatur


int main() {
printf("Hello, world!");
return 0;
}
msg: .data
.asciiz "Hello, world!"
.text
.globl main
main: la $a0,msg
li $v0,4
syscall
jr $ra

Literatur

[BO10] Bryant, Randal E. und David R. O´Hallaron: *Computer Systems - A Programmer´s Perspective*.
Prentice Hall, 2010.

[HH16] Harris, David Money und Sarah L. Harris: *Digital Design and Computer Architecture, ARM® Edition*.
Morgan Kaufmann, 2016.