Vorlesungsmodul Systemprogrammierung 1
- VorlMod SysProg1 -

Matthias Ansorg

20. März 2003 bis 28. März 2005

Zusammenfassung

Studentische Mitschrift zur Vorlesung Systemprogrammierung 1 bei Dr.-Ing. Karl-Heinrich Schmidt (Sommersemester 2003) im Studiengang Informatik an der Fachhochschule Gießen-Friedberg. In den Übungen dieser Veranstaltung wurde bisher Turbo Assembler und Turbo Linker verwendet, nun MS Visual C++ 6.0 zur 32-Bit Assemblerprogrammierung. Die Gliederung dieser studentischen Mitschrift ist identisch mit [1].

- **Bezugsquelle:** Die vorliegende studentische Mitschrift steht im Internet zum Download bereit: Persönliche Homepage Matthias Ansorg http://matthias.ansorgs.de/InformatikDiplom/Modul.SysProg1.Schmidt/.
- **Lizenz:** Diese studentische Mitschrift ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der angegebenen Quellen zu beachten.
- **Korrekturen und Feedback:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg <mailto:matthias@ansorgs.de>.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX (graphisches Frontend zu LaTeX) unter Linux geschrieben und mit pdfLaTeXals pdf-Datei erstellt. Grafiken wurden mit dem Programm xfig unter Linux erstellt und als pdf-Dateien exportiert.
- **Dozent:** Lehrbeauftragter Dr.-Ing. Karl-Heinrich Schmidt. Nur stundenweise und nicht ganztags an der FH Gießen-Friedberg anzutreffen.
- **Verwendete Quellen:** Die angegebene Literatur beschreibt fast ausschließlich 16-Bit Programmierung. Die Befehle zur 32-Bit Programmierung, auf die in dieser Veranstaltung eingegangen wird, sind identisch, nur die Operanden haben ein vorangestelltes E (»extended«).
- **Klausur:**
  - Hilfsmittel: das Skript von Prof. Wüst und beliebig viele andere Referenzhandbücher. Nützlich ist [1, S. 55]. Alle Skripte und Bücher dürfen verwendet werden, auch eigene Mitschriften. Taschenrechner und Notebook und andere Elektronik sind nicht erlaubt. Aufgabensammlungen, alte Klausuren und andere Notizen sind ebenfalls erlaubt. Also: Papier in jeder Form.
  - Teilnahmevoraussetzung: zwei anerkannte Hausübungen
  - Die Klausur wird von K.-H. Schmidt gestellt.

- Die erlaubten Hilfsmittel hat K.-H. Schmidt noch nicht festgelegt.
- Die MMX-Erweiterungen des Pentium werden in der Veranstaltung behandelt.
- Die Klausur wird ähnlich den alten Klausuren von Prof. Wüst sein. Diese kann man also als Vorlage nutzen, sie müssen irgendwo im Internet zu finden sein. Schwerpunkte der Klausur:
  * Inline-Prozeduren mit Parameterübergabe über Stack und Rückgabe über EAX oder ST0.
  * Aufgaben zu den Basisbefehlen MOV, ADD, SUB, ROL, ROR, XOR usw.
  * Aufgaben zu Gleitkommaarithmetik.
  * Im Gegensatz zu Klausuren von Prof. Wüst werden stets 32bit-Register verwendet, jedoch kein Int 21h usw.. Ausgelassen werden aus dem Skript: Ein- und Ausgabe; Betriebssystemfunktionen; MMX-Einheit. Im Gegensatz zu Klausuren von Prof. Wüst werden weniger reine TurboAssembler-Prozeduren verlangt, eher C-kompatible Prozeduren und Inline-Assembler.
  * Aufgaben, in denen man eine Prozedur schreiben muss und dabei den Lösungsweg überlegen muss (etwa mit Bitoperationen). Eine solche Aufgabe, in der man den Lösungsweg selbst entwickeln muss statt nur einen vorgegebenen Algorithmus in Assembler zu implementieren hat, ist natürlich aufwendig zu korrigieren, wird deshalb wahrscheinlich nicht vorkommen.
  * Rechnungen in der Klausur sind grundsätzlich so gestellt, dass man sie im Kopf lösen kann. Divisor und Faktoren sind also Zweierpotenzen, so dass man Bitschieben zur Division und Multiplikation verwenden kann.
  * Tabelle zur Umwandlung von dezimalen, hexadezimalen und binären Zahlen als Hilfsmittel erstellen.

Inhaltsverzeichnis

1 Einführung **5**
- 1.1 Maschinencode und Assemblersprache . . . 5
- 1.2 Register und Flags des 80386 . . . 6
- 1.3 Ein erstes Programm in Assembler . . . 6

2 Organisation und Benutzung des Hauptspeichers **6**
- 2.1 Speichervariablen definieren . . . 6
- 2.2 16-Bit-Umgebungen: Der segmentierte Hauptspeicher . . . 6
- 2.3 31-Bit-Umgebungen: Der unsegmentierte Hauptspeicher . . . 6
- 2.4 Addressierungsarten . . . 6
- 2.5 Testfragen . . . 6

3 Daten transportieren **6**
- 3.1 Daten gleicher Bitbreite kopieren - MOV . . . 6
- 3.2 Daten austauschen - XCHG . . . 6
- 3.3 Daten in größere Register transportieren . . . 6
- 3.4 Bedingtes Setzen von Registern oder Speicherplätzen . . . 6
- 3.5 Testfragen . . . 6

4 Ein- und Ausgabe **7**

5 Betriebssystemaufrufe **7**
5.1 Allgemeines . . . 7
5.2 Ausführung von Betriebssystemaufrufen in Assembler . . . 7
5.3 Einige Nützliche Betriebssystemaufrufe . . . 7
5.4 Testfragen . . . 7

6 Bitverarbeitung **7**
6.1 Bitweise logische Befehle . . . 7
6.2 Schiebe- und Rotationsbefehle . . . 7
6.3 Einzelbit-Befehle . . . 7
6.4 Testfragen . . . 7

7 Sprungbefehle **7**
7.1 Unbedingter Sprungbefehl - JMP . . . 7
7.2 Bedingte Sprungbefehle . . . 7
7.3 Verzweigungen und Schleifen . . . 8
7.4 Die Loop-Befehle . . . 8
7.5 Testfragen . . . 8

8 Arithmetische Befehle **8**
8.1 Die Darstellung von ganzen Zahlen . . . 8
8.2 Addition und Subtraktion . . . 8
8.3 Multiplikation . . . 8
8.4 Division . . . 8
8.5 Vorzeichenumkehr: NEG . . . 8
8.6 Beispiel . . . 8
8.7 Testfragen . . . 8

9 Stack und Stackbefehle **8**
9.1 Stackorganisation . . . 8
9.2 Stacküberlauf . . . 9
9.3 Anwendungsbeispiele . . . 9
9.4 Testfragen . . . 9

10 Unterprogramme **9**

11 Die Gleitkommaeinheit **9**
11.1 Gleitkommazahlen . . . 9
11.2 Aufbau der Gleitkommaeinheit . . . 9
11.3 Befehlssatz . . . 9

12 Die MMX-Einheit **9**
12.1 SIMD, Sättigungsarithmetik und MAC-Befehle . . . 9
12.2 Register, Datenformate und Befehle . . . 9
12.3 Der PMADDWD-Befehl: Unterstützung der digitalen Signalverarbeitung . . . 9
12.4 Befehlsübersicht . . . 9

13 Die Schnittstelle zwischen Assembler und C/C++ **10**
13.1 Übersicht . . . 10
13.2 16-/32-Bit-Umgebungen . . . 10
13.3 Aufbau und Funktion des Stack . . . 10
13.4 Erzeugung von Assemblercode durch Compiler . . . 10
13.5 Steuerung der Kompilierung . . . 10
13.6 Einbindung von Assemblercode in C/C++-Programme . . . 10

14 Assemblerpraxis **10**
14.1 Der Zeichensatz . . . 10
14.2 Die DOS-Kommandozeile - zurück in die Steinzeit . . . 10
14.3 Assemblieren, Linken, Debuggen . . . 10
14.4 Ein Rahmenprogramm . . . 10

15 Lösungen zu den Testfragen **10**

16 Assemblerbefehle nach Gruppen **11**
16.1 Allgemeines . . . 11
16.2 Transportbefehle . . . 11
16.3 Logische Befehle . . . 11
16.4 Schiebe- und Rotationsbefehle . . . 11
16.5 Einzelbit-Befehle . . . 11
16.6 Arithmetische Befehle . . . 11
16.7 Stackbefehle . . . 11
16.8 Programmfluss-Steuerungsbefehle . . . 11
16.9 Stringbefehle . . . 11
16.10 Ein- und Ausgabebefehle (Input/Output) . . . 11
16.11 Schleifenbefehle . . . 11
16.12 Prozessorkontrollbefehle . . . 11

17 Übungsaufgaben und Lösungen **12**
17.1 Übungsblatt 2, Aufgabe 1 . . . 12
17.2 Übungsblatt 2, Aufgabe 2 . . . 13
17.3 Übungsblatt 4, Aufgabe 4 . . . 13
17.4 Übungsblatt 5, Aufgabe 1 . . . 13
17.5 Übungsblatt 5, Aufgabe 2 . . . 14
17.6 Übungsblatt 5, Aufgabe 3 . . . 14
17.7 Übungsblatt 7 . . . 14
17.8 Übungsblatt 8, Aufgabe 1 . . . 14
17.9 Übungsblatt 8, Aufgabe 2 . . . 16
17.10 Übungsblatt 8, Aufgabe 3 . . . 17
17.11 Klausur Wüst 1999-07, Aufgabe 1 . . . 17
17.12 Klausur Wüst 2002-07, Aufgabe 2 . . . 17

18 Assembler unter Linux **18**
18.1 Informationsquellen . . . 18
18.2 Programm-Grundgerüst . . . 18

Abbildungsverzeichnis

1 Einführung

1.1 Maschinencode und Assemblersprache

Was heißt Systemprogrammierung? Assemblerprogrammierung wird hauptsächlich für geschwindigkeitskritische Bereiche des Betriebssystems eingesetzt. Daher die Bezeichnung »Systemprogrammierung«, bezogen auf das Betriebssystem und nicht auf die Hardware. Andere Bezeichnungen sind »Maschinennahe Programmierung« und »Assemblerprogrammierung«. Eigenschaften:

- prozessororientierte und prozessorspezifische Programmierung
- maschinenoptimierte und nicht problemorientierte bzw. objektorientierte Art der Programmierung
- hardwarenah
- nicht portabel auf andere Prozessoren
- programmiert wird mit Maschinenbefehlen
- ein Assembler übernimmt die Umsetzung von Mnemonics in Bytecodes.
- zu jedem Prozessor wurde mindestens eine Assemblersprache spezifiziert; in dieser Veranstaltung wird die Assemblersprache der Prozessorfamilie Intel 80x86 und Pentium behandelt.

Wozu Assemblerprogrammierung?

- Sie kann nicht zum Lösen komplexer Probleme verwendet werden.
- Laufzeitoptimierung. Assemblerprogramme sind um etwa Faktor 10 schneller als ein mit C++ entwickeltes Programm. Etwa bei Videobearbeitung ist dies auch heute noch notwendig. Es gab stets Anwendungen, die den Rechner überlasten, und das wird sich auch in Zukunft nicht ändern. Natürlich wird heute kein Programm mehr in reinem Assembler programmiert, sondern nur die sehr häufig ausgeführten Programmteile werden zuerst in einer höheren Programmiersprache geschrieben und dann durch Assembler ersetzt.
- optimale Prozessorausnutzung.
- vollständige Kontrolle des Prozessors. Dies ist unter Multitasking-Betriebssystemen aufgrund der »hardware abstraction layer« nur eingschränkt möglich.
- kompakter Code. Kompakte und übersichtliche Beschreibung eines Problems.
- besseres Verständnis der Arbeitsweise eines Computers. Dies ist der Hauptgrund für diese Veranstaltung.

Nachteile der Assemblerprogrammierung

- große Programme werden schnell unübersichtlich.
- nicht portabel.
- der Pentium-Befehlssatz enthält viel historischen Ballast. Diese Abwärtskompatibilität ist für Softwarehersteller natürlich vorteilhaft, weil so auch die ältesten Programme auch auf den modernsten Prozessoren und unter den neuesten Betriebssystemen laufen können.

1.2 Register und Flags des 80386

Alle Register EAX, EBX, ECX, EDX können für beliebige Zwecke verwendet werden.

1.3 Ein erstes Programm in Assembler

2 Organisation und Benutzung des Hauptspeichers

2.1 Speichervariablen definieren

2.2 16-Bit-Umgebungen: Der segmentierte Hauptspeicher

2.3 31-Bit-Umgebungen: Der unsegmentierte Hauptspeicher

2.4 Addressierungsarten

2.5 Testfragen

3 Daten transportieren

3.1 Daten gleicher Bitbreite kopieren - MOV

3.2 Daten austauschen - XCHG

3.3 Daten in größere Register transportieren

3.4 Bedingtes Setzen von Registern oder Speicherplätzen

3.5 Testfragen

4 Ein- und Ausgabe

5 Betriebssystemaufrufe

5.1 Allgemeines

5.2 Ausführung von Betriebssystemaufrufen in Assembler

5.3 Einige Nützliche Betriebssystemaufrufe

5.4 Testfragen

6 Bitverarbeitung

6.1 Bitweise logische Befehle

6.2 Schiebe- und Rotationsbefehle

6.3 Einzelbit-Befehle

6.4 Testfragen

7 Sprungbefehle

7.1 Unbedingter Sprungbefehl - JMP

7.2 Bedingte Sprungbefehle

7.3 Verzweigungen und Schleifen

7.4 Die Loop-Befehle

7.5 Testfragen

8 Arithmetische Befehle

8.1 Die Darstellung von ganzen Zahlen

8.2 Addition und Subtraktion

8.3 Multiplikation

8.4 Division

8.5 Vorzeichenumkehr: NEG

8.6 Beispiel

8.7 Testfragen

9 Stack und Stackbefehle

9.1 Stackorganisation

9.2 Stacküberlauf

9.3 Anwendungsbeispiele

9.4 Testfragen

10 Unterprogramme

11 Die Gleitkommaeinheit

11.1 Gleitkommazahlen

11.2 Aufbau der Gleitkommaeinheit

11.3 Befehlssatz

12 Die MMX-Einheit

12.1 SIMD, Sättigungsarithmetik und MAC-Befehle

12.2 Register, Datenformate und Befehle

12.3 Der PMADDWD-Befehl: Unterstützung der digitalen Signalverarbeitung

12.4 Befehlsübersicht

13 Die Schnittstelle zwischen Assembler und C/C++

13.1 Übersicht

13.2 16-/32-Bit-Umgebungen

13.3 Aufbau und Funktion des Stack

13.4 Erzeugung von Assemblercode durch Compiler

13.5 Steuerung der Kompilierung

13.6 Einbindung von Assemblercode in C/C++-Programme

14 Assemblerpraxis

14.1 Der Zeichensatz

14.2 Die DOS-Kommandozeile - zurück in die Steinzeit

14.3 Assemblieren, Linken, Debuggen

14.4 Ein Rahmenprogramm

15 Lösungen zu den Testfragen

16 Assemblerbefehle nach Gruppen

16.1 Allgemeines

16.2 Transportbefehle

16.3 Logische Befehle

16.4 Schiebe- und Rotationsbefehle

16.5 Einzelbit-Befehle

16.6 Arithmetische Befehle

16.7 Stackbefehle

16.8 Programmfluss-Steuerungsbefehle

16.9 Stringbefehle

16.10 Ein- und Ausgabebefehle (Input/Output)

16.11 Schleifenbefehle

16.12 Prozessorkontrollbefehle

17 Übungsaufgaben und Lösungen

Die folgenden Aufgaben stammen von den Übungsblättern von Lehrbeauftragtem Dr.-Ing. Karl-Heinrich Schmidt aus dem Sommersemester 2003 [6].

17.1 Übungsblatt 2, Aufgabe 1

Das »Hallo Welt«-C-Programm wurde hier mit GNU gcc realisiert. Es ist in der beiligeneden Datei `AufgBl2.Lsg/HelloWorld.c` enthalten, das zugehörige hier analysierte Assemblerlisting in der beiliegenden Datei `AufgBl2.Lsg/HelloWorld.listing`.

Auch die geschweiften Klammern {} erzeugen Assemblercode. Wozu dient der? Die öffnende geschweifte Klammer meint eigentlich den Funktionsbeginn. main() wird dabei wie jede andere Funktion auch behandelt: zuerst den Basepointer der rufenden Funktion sichern und den bisherigen »top of stack« als eigenen »bottom of stack« verwenden. Dies ist notwendig, weil indirekte Adressierung nicht mit esp, wohl aber mit ebp möglich ist.

```
26:HelloWorld.c  **** int main() {
29 0000 55               pushl   %ebp
31 0001 89E5             movl    %esp, %ebp
```

Die nächsten beiden Zeilen ergeben vermutlich ».align 4«: die Adressen eventuell folgender lokaler Variable sollen durch 4 teilbar sein, um schnelleren Hauptspeicherzugriff zu gewährleisten:

```
33 0003 83EC08           subl    $8, %esp
35 0006 83E4F0           andl    $-16, %esp   ;letztes Byte in %esp loeschen
```

Platz schaffen für lokale Variable; da es keine lokalen Variablen gibt, wird auch kein Platz (»`$0`«) freigemacht. Zwar wird `%esp` nicht benötigt, um die lokalen Variablen anzusprechen, sondern nur `%ebp` und ein negativer Index - die Variablen müssen aber trotzdem im Stack liegen, weshalb `%esp` trotzdem verändert werden muss.

```
36 0009 B8000000         movl    $0, %eax
37 000e 29C4             subl    %eax, %esp
```

Die schließende Klammer meint eigentlich das Funktionsende:

```
29:HelloWorld.c  **** }
47 0025 C9               leave
48 0026 C3               ret
```

Was macht das Maschinenprogramm für den printf() Aufruf?

```
27:HelloWorld.c  ****   printf("Hello World\n");
```

Parameterablage: dies ist eine Aufgabe des rufenden Programms. Hier werden 12 Byte im Stack bereitgestellt, um die Textkonstante »Hello World\n« hineinzukopieren, und die Adresse dieser Textkonstanten. Das eigentliche Hineinkopieren geschieht wohl in `printf` selbst.

```
39 0010 83EC0C           subl    $12, %esp
40 0013 68000000         pushl   $.LC0
```

Anschließend wird die Funktion aufgerufen:

```
42 0018 E8FCFFFF          call    printf
```

Und schließlich werden die Parameter wieder vom Stack entfernt. Es sind 16 Byte, nämlich 12 Byte die in Zeile 39 reserviert wurden und 4 Byte die in Zeile 40 für die Adresse der Textkonstanten benötigt wurden.

```
43 001d 83C410            addl    $16, %esp
```

Wie wird `return 0;` umgesetzt? Der Rückgabewert wird im Register `%eax` zurückgegeben:

```
28:HelloWorld.c **** return 0;
45 0020 B8000000 movl $0, %eax
```

17.2 Übungsblatt 2, Aufgabe 2

Welche Probleme treten bei der Umsetzung von 16bit- in 32bit-Code auf? Vergleiche [1, Kap. 13.2].

- 16bit-Code verwendet ein segmentiertes Speichermodell, 32bit-Code verwendet ein flaches Speichermodell und benutzt die Segmentregister für andere Zwecke.
- Stackbefehle beziehen sich in 16bit-Code auf ein Maschinenwort von 16 Bit, in 32bit-Code auf ein Maschinenwort von 32 Bit.
- Betriebssystemanbindung unterscheidet sich: über Software-Interrupts in 16bit-Code, über Funktionsaufrufe in 32bit-Code.

Warum funktioniert der int- Befehl nicht? `int 21h` ist ein Software-Interrupt. Er benötigt einen korrekten Verweis auf die Interrupt-Service-Routine (ISR) in der Interrupttabelle. Ein solcher existiert in 32bit-Umgebungen nicht.

17.3 Übungsblatt 4, Aufgabe 4

Das Programm für GNU Assembler ist in der beiliegenden Datei `./AufgBl04.Lsg/Int2ASCII.s` enthalten.

17.4 Übungsblatt 5, Aufgabe 1

»Wählen Sie eine dieser Standardfunktionen aus (z.B. memcpy) und implementieren Sie diese als normale C Funktion mit ähnlichem Namen (z.B. MemCopy). Erstellen Sie dazu ein neues Microsoft Visual C++ Projekt (Win32-Konsolenanwendung).«
Die Lösung in Form eines Microsoft Visual C++ Projektes befindet sich in `./AufgBl05.Lsg/StringCompare_VisualC`

»Kompilieren Sie ihre C Funktion mit eingeschalteter Listing-Option und schauen Sie sich im Listing den vom Compiler erzeugten Assembler-Code an.« Der Assembler-Code befindet sich vollständig in `./AufgBl05.Lsg/Loesung.StringCompare.orig.asm`.

»Führen Sie Ihr Programm im Disassembler-Fenster schrittweise aus und erläutern Sie, was passiert.« Hier wird mit `./AufgBl05.Lsg/Loesung.StringCompare.annotated.asm` eine kommentierte Version des Assemblerprogramms zur Verfügung gestellt, was einer erläuterten Ausführung im Disassembler entspricht.

17.5 Übungsblatt 5, Aufgabe 2

»Erläutern Sie (in Stichworten), wie Microsoft versucht, die Laufzeit zu optimieren. Welche Fähigkeiten des Prozessors werden genutzt? Welche Voraussetzung sollte der Aufrufer beachten, damit dies gelingt?« Lösung ist in Form von Kommentaren zur Microsoft-Funktion enthalten in `./AufgBl05.Lsg/Microsoft.strcmp.annotated.asm`.

Für die Laufzeitoptimierung sind allein die häufig ausgeführten Teile wichtig, nur diese wurden daher kommentiert. In der Microsoft-Version ist dies das Assembler-Äquivalent zu

```
while( ! (ret = *src - *dst) && *dst)
  ++src, ++dst;
```

also der Codeabschnitt `dodwords`. Die Codeabschnitte `dopartial` und `doword` werden dagegen höchstens einmal zu Beginn ausgeführt. Da es hier allein um Assemblerprogrammierung gehen soll, wird dir unterschiedliche Effizienz der Algorithmen selbst nicht untersucht, sondern nur die unterschiedlich effiziente Art, den jeweiligen Algorithmus in Assembler zu implementieren.

17.6 Übungsblatt 5, Aufgabe 3

»Schätzen Sie, um wie viel die Microsoft Funktion schneller ist als Ihre Version. (Begründung)« Lösung enthalten in `./AufgBl05.Lsg/Microsoft.strcmp.annotated.asm`.

»Stoppen Sie die Ausführungsdauer Ihrer Funktion. [...] Rufen sie zum Vergleich die C-Standardfunktion auf. Stoppen Sie wieder die Zeit.« Das Programm zum Stoppen der Ausführungszeiten für beide Funktionen ist in `./AufgBl05.Lsg/Profiler.StringCompare-vs-strcmp.cpp` enthalten.

»War Ihre Schätzung richtig?« Ausgabe des Programms für hinreichend große Felder: Die Version von Microsoft ist etwa 4,5 mal schneller. Die Schätzung »10 mal schneller« ist wohl deshalb so ungenau, weil sie nur auf den Speichertransferzeiten basiert (und nicht auch auf den unterschiedlichen Ausführungszeiten der Algorithmik) und die Speichertransferzeiten nicht so ausschlaggebend wie gedacht sind da die Routine zum Messen der Ausführungszeit die Cache-Effekte nicht ausschließt, sondern einen Durchschnitt ermittelt.

17.7 Übungsblatt 7

17.8 Übungsblatt 8, Aufgabe 1

1. Wie werden die Variablen i1, i2, f1, f2, d1, d2 initalisiert? (Assembler-Befehle, Werte)

   i1 (Initialisierungswert 99)

   ```
   movl $99, -4(%ebp)
   ```

   i2 (Initialisierungswert -1)

   ```
   movl $-1, -8(%ebp)
   ```

f1 (Initialisierungswert $42C76666_{16}$)

```
movl $0x42c76666, -16(%ebp)
```

f2 (Initialisierungswert $BE99999A_{16}$)

```
movl $0xbe99999a, -20(%ebp)
```

d1 (Initialisierungswert in den niederwertigen 4 Byte −858993459, in den höherwertigen 4 Byte 1079569612)

```
movl $-858993459, -32(%ebp)
movl $1079569612, -28(%ebp)
```

d2 (Initialisierungswert in den niederwertigen 4 Byte −858993459, in den höherwertigen 4 Byte 1076677837)

```
movl $858993459, -40(%ebp)
movl $-1076677837, -36(%ebp)
```

2. Wie werden diese Variablen an die jeweilige Differenzfunktion übergeben?

   i1, i2 werden über den Stack übergeben, 1 `pushl` pro Variable

   f1, f2 werden über den Stack übergeben, 1 `pushl` pro Variable

   d1, d2 werden über den Stack übergeben, 2 `pushl` pro Variable

3. Wie werden die Parameter in der Differenzfunktion referenziert?

   idifferenz

   ```
   movl 12(%ebp), %edx
   movl 8(%ebp), %eax
   ```

   fdifferenz

   ```
   flds 8(%ebp)
   fsubs 12(%ebp)
   ```

   ddifferenz

   ```
   fldl -8(%ebp)
   fsubl -16(%ebp)
   ```

4. Mit welchen Befehlen wird die Subtraktion jeweils ausgeführt?

   idifferenz `subl`

   fdifferenz `fsubs`

   ddifferenz `fsubl`

5. Wie werden die Funktionswerte an den Aufrufer zurückgeliefert?

   idifferenz

   ```
   movl -4(%ebp), %eax
   ```

   fdifferenz

   ```
   movl -4(%ebp), %eax
   movl %eax, -8(%ebp)
   flds -8(%ebp)
   ```

ddifferenz

```
movl -24(%ebp), %eax
movl -20(%ebp), %edx
movl %eax, -32(%ebp)
movl %edx, -28(%ebp)
fldl -32(%ebp)
```

6. Und wie kopiert der Aufrufer diese in die Ergebnisvariablen?

   idifferenz

   ```
   movl %eax, -12(%ebp)
   ```

   fdifferenz

   ```
   fstps -24(%ebp)
   ```

   ddifferenz

   ```
   fstpl -48(%ebp)
   ```

7. Wie sehen die Gleitkomma-Werte f1, f2, f3, d1, d2, d3 in hexadezimaler Form aus? Ermittelt per Debugger bzw. direkt aus dem Assemblerlisting.

   f1 $42C76666_{16}$, war im Assemblerlisting in diesem Format gegeben.

   f2 $BE99999A_{16}$, war im Assemblerlisting in diesem Format gegeben.

   f3 $42C80000_{16}$

   d1 $4058eccccccccccd_{16}$

   d2 $42c800004058eccc_{16}$

   d3 4059000000000000_{16}

17.9 Übungsblatt 8, Aufgabe 2

Zugehöriger Quelltext in `Solution.PiValue.cc`.

ST(0)

Dezimal 3.14159265358979323851280895940618 62

Hexadezimal $4000C90FDAA22168C235_{16}$

d3

Dezimal 3.1415926535897931

Hexadezimal $400921fb54442d18_{16}$

Hexadezimal Interpretiert

Mantisse $921fb54442d18_{16}$

Exponent 400_{16}

Vorzeichen 0_{16}

f3

Dezimal 3.14159274

Hexadezimal $40490fdb_{16}$

Hexadezimal Interpretiert

Mantisse $490fdb_{16}$

Exponent 40_{16}

Vorzeichen 0_{16}

17.10 Übungsblatt 8, Aufgabe 3

float	binär als `Vorzeichen:Exponent:Mantisse`	hexadezimal
1.0	`0:0111111 1:0000000 00000000 00000000b`	`3f80 0000h`
2.0	`0:1000000 0:0000000 00000000 00000000b`	`4000 0000h`
5.0	`0:1000000 1:0100000 00000000 00000000b`	`40a0 0000h`
100.0	`0:1000010 1:1001000 00000000 00000000b`	`42c8 0000h`
256.0	`0:1000011 1:0000000 00000000 00000000b`	`4380 0000h`
0.2	`0:0111110 0:1001100 11001100 11001101b`	`3e4c cccdh`
0.125	`0:0111110 0:0000000 00000000 00000000b`	`3e00 0000h`
-0.5	`1:0111111 0:0000000 00000000 00000000b`	`bf00 0000h`

17.11 Klausur Wüst 1999-07, Aufgabe 1

Geben Sie bitte in hexadezimaler Schreibweise an, welchen Ingalt die Register AX, BX, CX und DX nach dem Durchlaufen des folgenden Programmstücks haben:

Als Schreibtischtest:

nach Anweisung ↓ ist:	AX	BX	CX	DX
xor dx,dx				0000h
mov ax,5	0005h			
mov bx,2		0002h		
div bx	0002h			0001h
div bx	8001h			0000h
mov cx,092A2h			92A2h	
sub cl,ch			9210h	
xor ch,ch			0010h	
or bx,cx		0012h		
rol bx,1		0024h		

17.12 Klausur Wüst 2002-07, Aufgabe 2

Schreiben Sie in Asssembler eine Prozedur, die zwei Bitmuster vergleicht und feststellt, welches die niedrigste Bitposition ist, auf der die beiden Bitmuster übereinstimmen. Beispiel: Bei den Bitmustern 00110011 und 01011100 ist die niedrigste übereinstimmende Bitposition die Bitposition 4. Die Schnittstelle soll sein:

Vor Aufruf:

EAX: erstes Bitmuster

EBX: zweites Bitmuster

Nach Aufruf:

ECX: Die niedrigste übereinstimmende Bitposition.

```
  xor  eax,ebx
  not  eax
  mov  ecx,0
  cmp  eax,0
  jz   ende
schleife:
  test eax,1
  jnz  ende
  shr  eax
  inc  ecx
  jmp  schleife
ende:
```

18 Assembler unter Linux

Unter Linux ergibt sich eine Möglichkeit, mit freier Software Assembler zu üben. Hier einige Hinweise, was man dazu braucht und wie man es verwendet.

18.1 Informationsquellen

Siehe [7], [8], [9], [10].

18.2 Programm-Grundgerüst

In der beiliegenden Datei `LinuxAsm.HelloWorld.s` ein einfaches HelloWorld-Programm als Grundgerüst für eigene Programme. Ähnlich dem in [7, 6.2. Hello, world!] (siehe hier für weitere Informationen . . .), jedoch angepasst auf Verwendung von Intel-Syntax statt AT&T-Syntax in GAS.

Literatur

[1] Prof. Dr. Klaus Wüst: »Die Assemblersprache der intel 80x86-Prozessoren«. Ein sehr empfehlenswertes Skript von einem Professor der FH Gießen-Friedberg. Diese Veranstaltung orientiert sich hauptsächlich hieran, weicht davon jedoch zugunsten von 32-Bit-Programmierung ab. Wer sich die Veranstaltung selbst erarbeiten will, sollte sich hieran orientieren. Bezugsquelle: für 2,50EUR zu kaufen in der Veranstaltung Systemprogrammierung 1 oder Download auf der Homepage von Prof. Dr. Wüst :: Maschinennahe Programmierung http://homepages.fh-giessen.de/~hg6458/mprog.html. Auch referenziert auf [6].

[2] »IA-32 Intel Architektur Software Developer's Manual«. Download unter http://www.intel.com/design/pentium4/manuals oder ftp://download.intel.com/design/pentium4/manuals.

[3] »Microsoft Visual C++ Assembler (Inline) Topics«. Ein Kapitel aus dem Microsoft Developers Network (MSDN), dem online-Handbuch im Microsoft Visual C++ Developer Studio. Quelle: http://homepages.fh-giessen.de/~hg13025/msvc_asm.html.

[4] Rainer Backer: »Programmiersprache Assembler«; rororo Verlag. Es befasst sich ausschließlich mit den Prozessoren 8086 bis 80486. Es ist in einigen Exemplaren in der Bibliothek der FH Gießen-Friedberg vorhanden.

[5] Wolfgang Link: »Assembler Programmierung«; Franzis Verlag.

[6] Homepage von Lehrbeauftragtem Dr.-Ing. K.-H. Schmidt http://homepages.fh-giessen.de/~13025. Hier stehen alle Übungsblätter und wichtige Ankündigungen zur Veranstaltung »Systemprogrammierung 1« zur Verfügung, außerdem auch [1].

[7] Konstantin Boldyshev und Francois-Rene Rideau: »Linux Assembly HOWTO«. Wohl Teil des Linux Documentation Project. Mit vielen Linux-Distributionen standardmäßig mitinstalliert, etwa unter `/usr/share/doc/howto/en/html/Assembly-HOWTO/index.html`.

[8] »2.12, 2.13 vs previous versions:« Einige Anmerkungen zum GNU-Assembler GAS. Quelle http://www.lxhp.in-berlin.de/lhpas86.html.

[9] Dokumentation zum GNU-Assembler GAS. Wenn GAS installiert wurde, ist diese Dokumentation als TEX-Info-Manual »as« erreichbar, etwa mit »info as«.

[10] H.-Peter Recktenwald: »Reference to Linux 2.{2,4} System Calls for Assembly Level Access«. Quelle http://www.lxhp.in-berlin.de/lhpsyscal.html.

[11] Martin Müller: »Oft geschriebene Programme in Maschinennahe Programmierung bei Prof. Dr. K. Wüst«; Version vom 26.6.2002. http://homepages.fh-giessen.de/~hg11474/dateien/Assembler/MaschProg_-_Oft_geschriebene_Programme.pdf. Der Link wurde vom Autor großzügigerweise zur Verfügung gestellt.

[12] Klausuren von Prof. Dr. Wüst. Dateinamen etwa Asmkl702.pdf, Amkl799.pdf.