Grundlagen der Informatik III

Wintersemester 2010/2011

Wolfgang Heenes, Patrik Schmittat

TECHNISCHE UNIVERSITÄT DARMSTADT

1. Aufgabenblatt

01.11.2010

Hinweis: Der Schnelltest und die Aufgaben sollen in den Übungsgruppen bearbeitet werden. Die Hausaufgaben sind in der Kalenderwoche 45 (8.11. bis 12.11.) bei den Tutoren in **physikalischer Form** (handschriftlich oder gedruckt) abzugeben. Bei allen Abgaben ist der Name des Tutors und die Übungsgruppe deutlich anzugeben. Bei Teamabgaben wird nur eine Lösung eingereicht, die alle Namen der Teammitglieder enthält.

Aufgabe 1: Schnelltest

Fragen	Antworten
1. Welche Komponenten beschreiben eine Von-Neumann-Architektur?	☐ Rechenwerk ☐ Steuerwerk ☐ Netzwerkkarte ☐ Tastatur ☐ Vereinigter Daten- und Programmspeicher
2. Welches sind Bestandteile eines klassischen Betriebssystems?	☐ BIOS (Basic Input Output System) ☐ Speicherverwaltung ☐ Speichercontroller ☐ Prozessverwaltung ☐ Graphische Benutzeroberfläche
3. Was sind die typischen Merkmale von Programmierung in Assembler-Sprachen?	☐ Geringes Abstraktionsniveau ☐ Gute Wartbarkeit ☐ Hohe Effizienz ☐ I. A. leichte Portabilität ☐ Typsicherheit
4. Welche Aussagen zu Compiler, Assembler, Linker und Lader sind zutreffend?	☐ Der Präprozessor übersetzt den Programmcode in eine andere Sprache. ☐ Unter einem Compiler versteht man allgemein ein Programm, das aus dem Code einer höheren Programmiersprache äquivalenten Programmcode einer niederen Sprache erzeugt. ☐ Der Assembler erzeugt als Ausgabe nicht immer Maschinencode. ☐ Der Linker verbindet mehrere Objekt-Programme zu einem ausführbaren Programm. ☐ Der Lader löst symbolische Referenzen im Programm zu dynamisch gebundenen Bibliotheken vor der Ausführung auf.

Aufgabe 2: What-Do-I-Do?

Gegeben ist folgendes Java-Programm:

```
public class whatdoido {
    public static void main(String[] args) {
        short n = Short.MAX_VALUE;
        for (short i = 0; i <= n; i++) {
            System.out.println("i ist: " + i);
        }
    }
}
```

Was passiert bei der Ausführung des Programms? Wodurch werden die Probleme verursacht.

Aufgabe 3: Registersatz/Programmiermodell

Beschreiben Sie kurz den Registersatz, der Ihnen bei der Intel IA32-Architektur zur Verfügung steht.

Aufgabe 4: Grundrechenarten

In dieser Aufgabe sollen die nachfolgenden Formeln in äquivalenten Assembler-Code umgewandelt werden. Dabei steht der Operator := für eine Zuweisung, die Variablen a, b, c sind in den Registern **%eax**, **%ebx**, **%ecx** abgelegt. Eine Überlaufbehandlung müssen Sie nicht durchführen und beachten Sie jedoch, dass es sich bei allen Werten und Variablen um **long-Werte** handelt. Als Assembler-Syntax können Sie das ATT-Format oder Intel-Format verwenden.

a) $a := 23$

b) $a := a + b$

c) $c := a - b$

d) $a := 7 \cdot b$

e) $a := b^4 + (c+1) \cdot b^2 + 1$

Aufgabe 5: Maschinensprache vs. Hochsprache

a) Wann ist es sinnvoll, Programme direkt in Assembler (statt in C oder JAVA) zu schreiben?

b) Nennen Sie einige Vorteile einer Hochsprache wie C gegenüber Assembler.

Aufgabe 6: Beschränkter Befehlssatz

Für diese Aufgabe stehen nur die folgenden Befehle zur Verfügung:

xor, **or**, **and**, **shl**, **shr**, **sal**, **sar**.

Sie können nicht davon ausgehen, dass ungenutzte Register auf Null gesetzt sind. Daher muss Sorge getragen werden, dass die verwendeten Register einen definierten Wert besitzen. Beachten Sie, dass es sich bei allen Werten und Variablen um **long-Werte** handelt.

a) `a = 42`

b) `a = b`

c) `a = (a - (a mod 16)) / 16`

d) `a = a mod 32`

e) `a = 4 * a`

f) `a = 2 ^ b` (es gilt: $b \geq 0$)

g) `a = b * 5` (Sie dürfen zusätzlich **add** verwenden)

Hausaufgabe 1: Reverse-Engineering (4 Punkte)

Im Folgenden sind zwei Assembler-Programme gegeben.

```
.data
a: .long 12
b: .long 3
c: .long 2
d: .long 12
e: .long 12

.text
movl a, %eax
movl b, %ebx
movl d, %ecx
addl $2, %eax
addl $1, %ebx
addl e, %ecx
div c
```

```
.data
a: .long 12
b: .long 3
c: .long 2
d: .long 12
e: .long 12

.text
movl a, %eax
movl b, %ebx
movl d, %ecx
movl e, %edx
addl $2, %eax
incl %ebx
addl %edx, %ecx
shrl $1, %eax
```

a) Geben Sie die Registerinhalte nach dem Durchlauf aller Befehle an.

b) Vergleichen Sie die beiden Assembler-Programme bezüglich Ihrer Ausführungszeit (angegeben in Takten).

```
movl Quelle, Ziel
movl Reg, Reg       : 2 Takte
movl Reg, Speicher  : 9 Takte
movl Speicher, Reg  : 8 Takte
movl Wert, Reg      : 4 Takte
movl Wert, Speicher : 10 Takte

addl Quelle, Ziel
addl Reg, Reg       : 3 Takte
addl Reg, Speicher  : 16 Takte
addl Speicher, Reg  : 9 Takte
addl Wert, Reg      : 4 Takte

div Quelle
div Reg             : 85 Takte
div Speicher        : 159 Takte

incl Quelle
incl Reg            : 3 Takte
incl Speicher       : 15 Takte

shrl n, Ziel
shrl 1, Reg         : 2 Takte
shrl %cl, Reg       : 8 + 4 * n Takte
shrl 1, Speicher    : 15 Takte
shrl %cl, Speicher  : 20 + 4 * n Takte
```

c) Mit welchem Befehl kann man eine Mulitiplikation mit dem Faktor zwei schnell ausführen.

Hausaufgabe 2: Parity-Bit (6 Punkte)

In der Netzwerktechnik werden oft verschiedene Arten von Parity-Bits genutzt, um sicher zu gehen, dass die Datenübertragung einwandfrei funktioniert hat. Das Parity-Bit in seiner einfachsten Form ist die Vervollständigung einer Bitfolge auf eine gerade Anzahl von 1er durch anfügen eines Bits. Als Beispiel wird die Bitfolge `1011` auf `1011 1` und `0110` auf `0110 0` erweitert.

Schreiben Sie ein Assemblerprogramm, das eine gegebene 4-Bit Zahl in **`%eax`** um das oben beschriebene Parity-Bit erweitert und das Ergebnis wieder in **`%eax`** ablegt. Sprünge oder konditionale Befehle stehen dabei nicht zur Verfügung. Kommentieren Sie Ihren Code.

Hinweis: Wenn Sie Ihren Algorithmus testen möchten, können Sie folgende Codefragmente nutzen, um die Werte eines Registers auf der Konsole auszugeben.

```
.data
intout:
    .string "Wert␣%d\n"

.text
.globl main

main:
# Wert fuer die Berechnung
movl $13, %eax # 13 == 0b1101

# ... Hier Ihren Code einfuegen ...

# Wert im %eax ausgeben
pushl %eax
pushl $intout
call printf

# Exit
movl $1, %eax
int $0x80
```

Wenn Sie Ihr Programm in der Datei parity.asm abspeichern, können Sie im RBG-Pool folgende Befehlsfolge zum Assemblieren und Linken nutzen.

```
yasm -f elf -p gas parity.asm
gcc -melf_i386 -o parity parity.o
./parity
```

Hinweise zu einem Shell-Skript finden Sie unter `http://d120.de/forum/viewtopic.php?f=179&t=20777`.