Autor: Sebastian Sossalla

Disclaimer: Das sind Lösungsvorschläge, keine Musterlösungen.

Aufgabe 1: Adressierungsarten

1a) Erklären Sie den Begriff „effektive Adresse“! (2 Punkte)

Siehe Klausur blabla

1b) Wann wird ein Programmierer bzw. ein Compiler beim Erzeugen von Assembler-Code die effektive Adresse direkt über bestimmte Adressierungsarten, wann über den `lea`-Befehl und wann über explizite Arithmetik-Befehle berechnen lassen? (3 Punkte)

Vermutung:
Wenn die Basis-Adresse des Speicherwertes zur Übersetzungszeit bekannt ist, wird die effektive Adresse primär über Adressierungsarten angesprochen. Der Lea-Befehl wird eingesetzt, falls die Basis-Adresse zum Beispiel als Parameter zum Beispiel auf dem Stack oder auf einem Register übergeben wird. Und die effektive Adresse zum Berechnet werden bei Programmier-Konstrukten wo Zugriff stattfinden soll, (z.B. c = array[i][k]) für den es keine eigene Adressierungsart gibt. Dann müssen die Adressen über arithmetische Befehle berechnet werden.

Aufgabe 2: Speicher (7 Punkte)

Ein Kunden-Datensatz bestehe aus folgenden Daten:

```
struct data {
     char name[21];
     long value;
     short id;
};
```

2a) Wieviel Platz braucht eine Datenbank bestehend aus 2048 derartigen Datensätzen auf einem 32-Bit-Rechner mit korrektem Alignment (char: 1 Byte, short: 2 Byte, long: 4 Byte)? (3 Punkte)

1234	5678	9 10 11 12	13 14 15 16	17 18 19 20	21 d d d	value	id d d		

2048 = 2 * 2^10 * (32) = 64 * 2^10 = 64 Kb

2b) Wie könnte man die Daten so umordnen, dass die Datenbank möglichst wenig Platz benötigt, alle Daten aber trotzdem noch passendes Alignment besitzen? (4 Punkte)

```
struct data {
     long value;
     short id;
     char name[21];
```

```
    char dummy[0];
};
```

value	id cc	cccc	cccc	cccc	cccc	cccd		

Aufgabe 3 spar' ich mir jetzt. Schon zu genüge den Mist gemacht...

Aufgabe 4: Unterprogramm / Stack (20 Punkte)

Der untenstehende Code wurde von einem Compiler aus einem Hochsprachen-Code generiert. Wie könnte die ursprüngliche Hochsprachen-Funktion ausgesehen haben?

```
func:
     subl $24,%esp
     pushl %ebx
     movl 32(%esp),%ebx
     cmpl $1,%ebx
     jle .L3
     addl $-12,%esp
     leal -1(%ebx),%eax
     pushl %eax
     call func
     imull %eax,%ebx
     movl %ebx,%eax
     addl $16,%esp
     jmp .L2
.L3:
     movl $1,%eax
.L2:
     popl %ebx
     addl $24,%esp
     ret
```

Aufgabe 5: Stack / Position-Independent-Code (8 Punkte)

Im Source-Code des OpenBSD-Betriebssystems für i386-kompatible CPUs findet sich folgender Assembler-Code-Teil:

```
...
pushl $begin
ret
begin:
...
```

5a) Was passiert bei der Ausführung dieses Code-Fragments? Begründen Sie Ihre Antwort! (3 Punkte)

Die Adresse, an der das Label *begin* steht, wird auf den Stack gelegt. Diese verwendet der ret-Befehl um dort hinzuspringen (und entfernt die Adresse wieder vom Stack durch anheben des Stack-Pointers).

5b) Warum könnte obiger Code überhaupt notwendig sein (Hinweis: Position-Independent-Code)? (5 Punkte)

Würde man einfach nur jmp begin schreiben, würde der Compiler einen Position-Independent-Code erzeugen und einfach nur einen Offset zur momentanen Adresse beim übersetzen generieren. Da man aber die Adresse auf den Stack pushed, und danach einen ret-Befehl ausführt, wird beim Übersetzen Dependend-Code erzeugt. Also die absolute Adresse wird verwendet. D.H. man kann nach dem 'begin:' Position-Dependent Code haben. Was man davor hatte, kann sowohl Position-Independent als auch Position-Dependent Code gewesen sein (auch wenn letzteres wenig Sinn macht, weil es dann auch ein normales 'jmp begin' getan hätte). Das Ganze ist sozusagen eine Schleuße zwischen Position-Independent Code und Position-Dependent Code.

Aufgabe 6: Hardware / ALU (5 Punkte)

Eine i386-kompatible CPU kennt einen `cbw`-Befehl (Convert Byte to Word). Dieser Befehl konvertiert eine 8-Bit-Signed-Integer-Zahl X in eine 16-Bit-Signed-Integer-Zahl Y.
Wie kann dieser Befehl in Hardware implementiert werden? (Hinweis: für die Implementierung werden keine Gatter gebraucht!)

Lösung:

Das ist auch wirklich kein Problem: Man lässt einfach die hinteren 7 Bit durchrauschen, und das 8. Bit wird an Ausgang 8-16 gelegt. Damit bleibt das vorzeichen erhalten!