64-041 Übung Rechnerstrukturen

Aufgabenblatt 13 Ausgabe: 17.01., Abgabe: 24.01. 12:00

Gruppe	
Name(n)	Matrikelnummer(n)

Aufgabe 13.1 (Punkte 5+5+5+5+5+5+5+5+5)

Adressierungsarten: Kreuzen Sie bitte an, welche der unten angebenen Adressierungsarten Sie den folgenden Befehle zuordnen würden. Es ist auch mehr als ein Kreuz pro Befehl möglich. Konzentrieren Sie sich dabei mehr auf die beteiligten Datentransfers als auf die Codierung der Befehle.

Bemerkung: Die Befehle **call** und **jmp** sind im x86-Befehlsformat so codiert, dass nicht die absolute Adresse, sondern der Offset zum aktuellen Stand des Programmzählers im Befehlswort steht.

Befehl	unmittelbar	Register	Direkt	Indirekt	Indiziert	PC+Offset
`movl %esp, %ebp`						
`movl 8(%ebp), %edx`						
`addl $1, %edx`						
`pushl %edx`						
`jmp ende`						
`call fu1`						
`popl %ebp`						
`ret`						
`leal 4(%ebx,%eax,2), %edx`						

Aufgabe 13.2 (Punkte 5+5+5+5+5+5+5)

Flags: Fast alle Prozessoren haben ein Carry-Flag und ein Overflow-Flag. Zur Erinnerung:
Das Carry-Flag wird bei einer arithmetischen Operation gesetzt, wenn sich in der höchstwertigsten Stelle ein Übertrag ergibt.
Das Overflow-Flag wird gesetzt, wenn das Ergebnis der Operation das "falsche" Vorzeichen hat. Z.B., wenn die Addition zweier positiver Zahlen ein negatives Ergebnis liefert.

Kreuzen Sie an, welches Flag bei Ausführung der folgenden Operationen gesetzt werden würde.

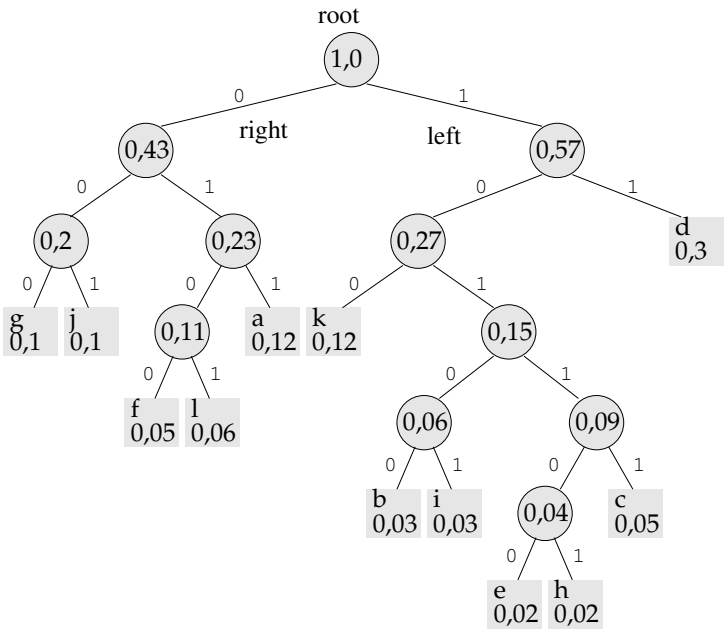
Operation	Inhalt %eax	Inhalt %ebx	CF	OF
`addl %eax, %ebx`	`0x00000001`	`0x00000002`		
`addl %eax, %ebx`	`0x00000001`	`0xFFFFFFFF`		
`addl %eax, %ebx`	`0x00000002`	`0x7FFFFFFF`		
`addl %eax, %ebx`	`0x80000000`	`0xFFFFFFFE`		
`subl %eax, %ebx`	`0x00000002`	`0x00000001`		
`subl %eax, %ebx`	`0x00000001`	`0x00000002`		
`subl %eax, %ebx`	`0x00000001`	`0x80000000`		

Aufgabe 13.3 (Punkte 20)

Datenstrukturen und Rekursion In Aufgabe 6.3 hatten Sie sich für die folgenden 12 Symbole a_i mit den Wahrscheinlichkeiten $p(a_i)$

a_i	a	b	c	d	e	f	g	h	i	j	k	l
$p(a_i)$	0,12	0,03	0,05	0,3	0,02	0,05	0,1	0,02	0,03	0,1	0,12	0,06

eine Huffman-Codierung überlegt. Das führte zu einem Baum der Form



Will man einen solchen Baum in einer Programmiersprache (hier C) aufbauen und damit arbeiten, braucht man zunächst einmal eine geeignete Datenstruktur für die einzelnen Knoten des Baums. Eine Möglichkeit von vielen ist die folgende:

```
struct  treenode {
    char    letter;
    short   probability;
    struct  treenode *left;
    struct  treenode *right;
};
```

Dabei ist:
letter das codierte Symbol (dargestellt als Buchstabe) in einem der Endknoten oder 0, wenn es sich um einen Knoten handelt, von dem weitere Zweige abgehen.
probability die Wahrscheinlichkeit (die Zahl, die in obigem Bild in jedem der Knoten steht) multipliziert mit 100 als 16 Bit Integer.
left, right Zeiger auf die nächsten Knoten im Baum oder NULL, wenn es sich um einen Endknoten handelt.

Die folgende Assembler-Routine geht rekursiv durch den Baum und gibt zu jedem Symbol das Symbol selbst, die Wahrscheinlichkeit und den entsprechenden Huffman-Code aus.

```
format:
.string     "%c  --  %2d --  %s\n"
#-------------------------------------------------------
# ausgabe(char *buffer, int ind, treenode *pointer)
#-------------------------------------------------------
ausgabe:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    movl    16(%ebp), %ebx
    movb    (%ebx), %al
    cmpb    $0, %al
    je      rekursion
    movl    12(%ebp), %eax
    addl    8(%ebp), %eax
    movb    $0, (%eax)
    movl    8(%ebp), %edx
    pushl   %edx
    movl    $0, %ecx
    movl    %ecx, %edx
    movw    2(%ebx), %cx
    pushl   %ecx
    movb    (%ebx), %dl
    pushl   %edx
    movl    $format, %eax
    pushl   %eax
    call    printf
    jmp     ende
rekursion:
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    movb   $48, (%eax)
    movl   16(%ebp), %eax
    movl   8(%eax), %eax
```

```
    movl   12(%ebp), %edx
    addl   $1, %edx
    pushl  %eax
    pushl  %edx
    movl   8(%ebp), %eax
    pushl  %eax
    call   ausgabe
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    movb   $49, (%eax)
    movl   16(%ebp), %eax
    movl   4(%eax), %eax
    movl   12(%ebp), %edx
    addl   $1, %edx
    pushl  %eax
    pushl  %edx
    movl   8(%ebp), %eax
    pushl  %eax
    call   ausgabe
ende:
    popl    %ebx
    movl    %ebp, %esp
    popl    %ebp
    ret
```

Kommentieren Sie die Befehle obiger Assembler-Routine.

Dazu noch einige Hinweise:
Im Parameter *Buffer* der Routine wird sukzessive der Huffman-Code aufgebaut. Der Parameter *ind* gibt dabei den nächsten freien Index in *buffer*. Wenn ein Endknoten erreicht ist, wird hier eine 0 (wirklich eine 0 und nicht der ASCII-Character für die Ziffer 0!) eingetragen, um das Ende des Strings zu markieren. Das braucht C, um den String dann ausgeben zu können. Sonst wird 48 als ASCII-Code für die Ziffer 0 bzw. 49 für die Ziffer 1 eingetragen.
Wenn *adr* die Adresse eines Knotens vom Typ *treenode* (s.o.) ist, dann ist *adr* die Adresse der Komponente *letter*, *adr+2* die Adresse der Komponente *probability*, *adr+4* bzw. *adr+8* die Adressen der Komponenten *left* und *right*.
Aufgerufen wird die Routine im Hauptprogramm mit *ausgabe(buffer, 0, root)*, wobei *root* die Wurzel des Baums ist.

Den Source-Code für die Assembler-Routine *ausgabe* und ein Test-Programm, das den Baum aufbaut und *ausgabe* aufruft, finden Sie zum Ausprobieren auf der Übungs-Webseite. Das Programm erzeugt dabei einen etwas anderen Baum als den oben angebenen, aber das sollte keine Rolle spielen. Wenn man den „richtigen“ möchte, kann man die Aufrufe von *init* und *make_huff* auskommentieren und dafür das, was auskommentiert ist, wieder in das Programm aufnehmen.